MP/M-86™

OPERATING SYSTEM PROGRAMMER'S GUIDE

DIGITAL RESEARCH®

MP/M-86 T.M.

Operating System

PROGRAMMER'S GUIDE

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

COPYRIGHT

DISCLAIMER

TRADEMARKS

The "MP/M-86 Programmer's Guide" was prepared using the Digital Research TEX Text Formatter and printed in the United States of America by Commercial Press / Monterey.

```
*****************************************
*  First Printing:   September 1981     *
*****************************************
```

FOREWORD

MP/M-86™ is a multi-user operating system for microcomputers that use the Intel 8086, 8088, or compatible microprocessor. It will support multi-terminal access with multi-programming at each terminal. The minimum hardware environment for MP/M-86 must include an 8086 or similar processor, 64K bytes of random access memory (RAM), a system console, and a real-time clock. A typical MP/M-86 kernel occupies less than 32K bytes.

This manual describes the programming interface to MP/M-86. Sections 1 through 6 describe the modules that comprise the operating system, the manner in which MP/M-86 monitors running processes, as well as detailed descriptions of all the system entry points.

Section 7 contains a complete description of the Digital Research assembler ASM-86™ and the various options that can be invoked with it. One of these options controls the hexadecimal output format. ASM-86 can generate 8086 machine code in either Intel or Digital Research format. Appendix A describes these formats.

Section 8 discusses the elements of ASM-86 assembly language. It defines ASM-86's character set, constants, variables, identifiers, operators, expressions, and statements.

Section 9 discusses the ASM-86 housekeeping functions such as requesting conditional assembly, including multiple source files, and controlling the format of the listing printout.

Section 10 summarizes the 8086 instruction mnemonics accepted by ASM-86. These mnemonics are the same as those used by the Intel assembler except for four instructions: the intra-segment short jump, and inter-segment jump, return and call instructions. Appendix B summarizes these differences.

Section 11 discusses the code-macro facilities of ASM-86, including code-macro definition, specifiers and modifiers as well as nine special code-macro directives. This information is also summarized in Appendix H.

Section 12 discusses DDT-86™, the interactive debugging program, which allows the user to test and debug programs in the 8086 environment. The section includes a sample debugging session.

This manual is not intended as a tutorial. Therefore, familiarity with the material covered in the User's Guide and with processor architecture and assembly language in general is assumed.

TABLE OF CONTENTS

TABLE OF CONTENTS

(continued)

TABLE OF CONTENTS

(continued)

TABLE OF CONTENTS

(continued)

TABLE OF CONTENTS

(continued)

TABLE OF CONTENTS

(continued)

APPENDIXES

SECTION 1

MP/M-86 SYSTEM OVERVIEW

1.1 Introduction

MP/M-86 is a microcomputer operating system that supports multiple terminals with multi-programming at each terminal. MP/M-86 is compatible with the single-user operating system, CP/M-86™. In addition, the system functions used by MP/M-86 to control the multi-programming environment are available to application programs. As a result, MP/M-86 supports extended features such as communication between and synchronization of independently running processes.

Under MP/M-86, there is an important distinction between a program and a process. A program is simply a block of code residing somewhere in memory or on disk; it is essentially static. A process on the other hand, is dynamic, and can be thought of as a "logical machine" that not only executes the program's code, but also executes code in the operating system. When MP/M-86 loads a program, it also creates a process that is associated with the loaded program. Subsequently, it is the process, rather than the program that controls all access to the system's resources. Thus, MP/M-86 monitors the process, not the program. This distinction is a subtle one, but vital to understanding the operation of the system as a whole.

Processes running under MP/M-86 fall into two categories: transient processes and MP/M-86 system processes (including Resident System Processes). The first category consists processes that run absolute memory images of programs the system loads from disk into available memory partitions.

The second category consists of MP/M-86 system processes that perform operating system tasks. For example, the IDLE process is a pre-defined process that does not perform any task but gives the system a process to execute when there are no other processes ready to run.

Resident System Processes (RSPs) are those processes that can be integrated into MP/M-86 during system generation, thus becoming a part of the system. For example, the TERMINAL MESSAGE PROCESS (TMP), is the system process that provides command line support for system consoles under MP/M-86. With RSPs, users can write custom processes and include them in the system along with those supplied with MP/M-86 (see Section 1.8 and Section 5). Note: All processes running under MP/M-86 compete for the CPU and other system resources on a priority basis under control of the Real-Time Monitor.

The following list briefly summarizes MP/M-86's capabilities.

- Multi-terminal support. MP/M-86 supports up to 254 character I/O devices. These include consoles and list devices. Although there is no set restriction on the number of devices specified during system generation, a typical number of system consoles would be 4 to 16. Also, under MP/M-86 a single process can access multiple terminals.
- Multi-programming at each terminal. Any system console can initiate multiple programs. In addition, once a process is initiated, it can can generate subprocesses.
- Inter-process communication, synchronization, and mutual exclusion. These functions are provided by system queues.
- Logical interrupt mechanism using flags. This allows MP/M-86 to interface with any physical interrupt structure.
- System timing functions. These functions enable processes running under MP/M-86 to compute elapsed times, delay execution for specified intervals, and to access and set the current date and time.
- User-selected options at system generation time. The available options include the number of system consoles and list devices, the number, size, and location of memory partitions, and the maximum number of locked files that can be opened on the system at one time. Also, the user can select which RSPs to include with MP/M-86 during system generation.

Functionally, MP/M-86 is composed of several distinct modules. They are: the Supervisor (SUP), the Real-Time Monitor (RTM), the Memory Management module (MEM), the Character I/O module (CIO), the Basic Disk Operating System (BDOS), and the Extended I/O System (XIOS). The SUP module handles miscellaneous system functions such as returning the version number or the address of the System Data Area, and also calls other system functions when necessary. The RTM module monitors the execution of running processes and arbitrates conflicts for the system's resources. The MEM module allocates and frees memory upon demand from executing processes. The CIO module handles all character I/O for console and list devices in the system. The (BDOS) is the hardware-independent module that contains the logically invariant portion of the file system for MP/M-86. The BDOS file system is explained in detail in Section 2. The XIOS is the hardware-dependent module that defines MP/M-86's interface to a particular hardware environment. Although a sample XIOS is supplied by Digital Research, the XIOS is usually customized by an OEM or distributer of MP/M-86 to support the user's local hardware environment.

When MP/M-86 is configured for a single console and is executing a single program, its speed approximates that of CP/M-86. In environments where either multiple processes and/or users are running, the speed of each individual process is degraded in proportion to the amount of I/O and compute resources required. A process that performs a large amount of I/O in proportion to computing exhibits only minor speed degradation. This also applies to a process that performs a large amount of computing, but is running concurrently with other processes that are largely I/O-bound. On the other hand, significant speed degradation occurs in those environments where more than one compute-bound process is running.

1.2 Supervisor (SUP)

The Supervisor module (SUP) manages the interaction between transient processes and the other system modules, including future networking interfaces. All system function calls, whether they originate from a transient process or internally from another system module, go through a common table-driven function interface. The SUP module handles all system functions that call other system functions, such as the PROGRAM LOAD and CLI (COMMAND LINE INTERPRETER) functions.

1.3 Real-Time Monitor (RTM)

MP/M-86 is controlled by a real-time multi-tasking nucleus called the Real-Time Monitor (RTM). The RTM performs process dispatching, queue management, flag management, device polling, and system timing tasks. Many of the system functions used to perform these tasks can also be called by user programs.

1.3.1 Process Dispatching

Although MP/M-86 is a multi-processing operating system, at any given point in time, only one process has access to the CPU resource. Unless it is specifically written to communicate or synchronize execution with other processes, it runs unaware that other processes may be competing for the system's resources. Eventually, the system suspends the process from execution and allows another process to run.

The primary task of the RTM is transferring the CPU resource from one process to another. This task is called dispatching and is performed by a part of the RTM called the Dispatcher. Each process running under MP/M-86 is associated with two data structures called the Process Descriptor (PD) and the User Data Area (UDA). The Dispatcher uses these data structures to save and restore the current state of a running process. Each process in the system resides in one of three states: ready, running, and suspended. A ready process is one that is waiting for the CPU resource only. A running process is one that the CPU is currently executing. A suspended process is one that is waiting for some other system resource or a defined event.

A dispatch operation can be summarized as follows:

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

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

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

4) The process executes until a resource is needed, a resource is freed, or an interrupt occurs. At this point, a dispatch occurs, allowing another process to run. The system clock generates interrupts once every clock tick (approximately 16ms) thereby generating time slices for CPU-bound processes.

Only processes that are placed on the Ready List are eligible for selection during dispatch. By definition, a process is on the Ready List if it is waiting for the CPU resource only. Processes waiting for other system resources cannot execute until their resource requirements are satisfied. Under MP/M-86, a process is blocked from execution if it is waiting for:

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

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

MP/M-86 is a priority-driven system. This means that the Dispatcher selects the highest priority ready process and gives it the CPU resource. Processes with the same priority are "round-robin" scheduled. That is, they are given equal shares of the system's resources. With priority dispatching, control is never passed to a lower priority process if there is a higher priority process on the Ready List. Since high priority compute-bound processes tend to monopolize the CPU resource, it is advisable to lower their priority to avoid degrading overall system performance.

MP/M-86 requires at least one process run at all times. To ensure this, the system maintains the IDLE process on the Ready List so it can be dispatched if there are no other processes available. The IDLE process runs at a very low priority and is never blocked from execution. It does not perform any useful task, but simply gives the system a process to run when no other ready processes exist.

1.3.2 Queue Management

Queues perform several critical functions for processes running under MP/M-86. They are used for communicating messages between processes, for synchronizing process execution, and for mutual exclusion. Each system queue is composed of two parts: the Queue Descriptor, and the Queue Buffer. These are special data structures implemented in MP/M-86 as "memory files" that contain room for a specified number of fixed length messages. Like files, queues are made, opened, deleted, read from, and written to with appropriate system function calls. When a queue is created by the MAKE QUEUE function call, it is assigned an 8-character name that identifies the queue in all the other function calls. As the name implies, messages are read from a queue on a first-in, first-out basis.

A process can read messages from a queue or write messages to a queue in two ways: conditionally or unconditionally. If no messages exist in the queue when a conditional read is performed, or the queue is full when a conditional write is performed, the system returns an Error Code to the calling process. On the other hand, if a process performs an unconditional read operation from an empty queue, the system suspends the process from execution until another process writes a message to the queue.

When more than one process is waiting for a message, preference is given to the higher priority process. Conflicts involving processes with the same priority are resolved on a first-come first-serve basis.

Mutual exclusion queues are a special type of queue under MP/M-86. They contain one message of zero length and are typically assigned a name beginning with the upper-case letters, MX. In effect, a mutual exclusion queue is a binary semaphore. Mutual exclusion queues ensure that only one process has access to a resource at a time.

Access to a process protected by a mutual exclusion queue takes place as follows:

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

2) The process accesses the protected resource.

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

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

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

1.3.3 System Timing Functions

MP/M-86's system timing functions include keeping the time of day, and delaying the execution of a process for a specified period of time. An internal process called CLOCK, provides the time of day for the system. This process issues FLAG WAIT calls on the system's "one second" flag, Flag 2. When the XIOS Interrupt Handler sets this flag, it initiates the CLOCK process which then increments the internal time and date. Subsequently, the CLOCK process makes another FLAG WAIT call and suspends itself until the flag is set again. MP/M-86 provides functions that allow the user to set and access the internal date and time. In addition, the file system uses the internal time and date to record when a file is updated, created, or last accessed.

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

1.4 Memory Module (MEM)

The Memory Module handles all memory management functions. MP/M-86 2.0 supports an extended, fixed partition model of memory management. In practice, the exact method that the operating system uses to allocate and free memory is transparent to the programmer. In fact, the programmer should take care to write code that is independent of the memory management model by using only the MP/M-86 system functions as described in Section 6. If the system functions are not used, incompatibilty may result since future versions of MP/M-86 may support different versions of the Memory module depending on the classes of memory management hardware that are available.

1.5 Character I/O module (CIO)

The Character I/O module handles all console and list I/O. Under MP/M-86, every character I/O device is associated with a data structure called a Character Control Block (CCB). The CCB contains the current owner, the root of a linked list of Process Descriptors (PDs) that are waiting for access, line editing variables, and status information. CCBs reside in the CCB Table of the System Data Area. Each Process Descriptor contains the CCB Index of its default console and list device. Consoles are mapped such that CCB Index 0 corresponds to console 0. List device CCBs start after the console CCBs. The number of CCBs in the CCB Table is a system generation option, and must be large enough to include all the console and list devices supported in the XIOS.

1.6 Basic Disk Operating System (BDOS)

The MP/M-86 BDOS is an upward-compatible version of the single-user CP/M-86 BDOS. It handles file creation and deletion, file access, either sequential or random, and allocates and frees disk space. In most cases, CP/M-86 programs that make BDOS calls for I/O can run under MP/M-86 without modification. MP/M-86's BDOS is extended to provide support for multiple console and list devices. In addition, the file system is extended to provide services required in multi-user environments. Two major extensions to the file system are:

- File locking. Normally, files opened under MP/M-86 cannot be opened or deleted by other users. This feature prevents accidental conflicts with other users.

- Shared access to files. As a special option, independent users can open the same file in shared or unlocked mode. MP/M-86 supports record locking and unlocking commands for files opened in this mode, and protects files opened in shared mode from deletion by other users.

1.7 Extended Input/Output System (XIOS)

The XIOS module is similar to the CP/M-86 Basic Input/Output System (BIOS) module but is extended in several ways. Primitive functions such as console I/O are modified to support multiple consoles. Several new primitive functions support MP/M-86's additional features. Also, new facilities are added to eliminate wait loops. Refer to the MP/M-86 System Guide for a detailed description of the XIOS.

1.8 Resident System Processes

Resident System Processes are considered part of the operating system. The system generation utility, GENSYS, prompts the user to select which RSPs to include in the system. All RSPs selected are placed next to each other immediately following the System Data Area (SYSDAT). The MP/M-86 System Guide describes in greater detail the manner in which the operating system modules reside in memory.

RSPs are permanently system resident, residing within the Operating System area. Thus, if an RSP creates a queue or a subprocess, the Process Descriptor, Queue Descriptor, and Queue Buffer areas are usually used directly by the Operating System instead of copying them into system tables. The only time these areas are copied is when the data structures are actually outside the 64K address space of the SYSDAT module. This is because all pointers to these structures are relative to the SYSDAT segment address.

1.9 Transient Programs

Under MP/M-86, a transient program is one that is not system resident. That is, the system must load it from disk into an available memory partition every time it executes. The command file of a transient program is identified by a file type of CMD. When a user enters a command at the console, the operating system searches on disk for the appropriate CMD file which it then loads and initiates. MP/M-86 supports three different execution models for transient programs. These models are explained in detail in Section 3.

1.10 Resident Procedure Library (RPL)

MP/M-86 supports a special type of RSP called a Resident Procedure Library (RPL). RPLs provide a method of utilizing a block of code as a system resource. A Resident Procedure Library is set up by an RSP. For each library procedure, the process creates a queue with the name of the RPL and sends it a single 4-byte message containing the double-word address of the procedure (code) to be accessed. Once this is accomplished, the RSP terminates itself.

The RPL is accessed by through the Function 151, CALL RPL. This function opens the queue and reads the message to obtain the actual memory address of the procedure. It then executes a Far Call instruction to this address. Because only one message can reside in the queue, only one process can gain access to the procedure until the message is written back to the queue. Thus a process can determine whether or not the procedure is used concurrently or serially, by writing the message back to the queue just after entry, or just prior to return. Once the procedure completes its intended function, it executes a Far Return instruction back to the CALL RPL routine, and finally back to the calling process.

1.11 System Function Calling Conventions

Under MP/M-86, when a process makes a system function call, it uses the protocol shown in Table 1-1.

Table 1-1. Register Usage For System Function Calls

```
                ENTRY PARAMETERS
                ================

     +------------------------------------+
     | Register  CL: Function Number      |
     |           DL: Byte Parameter       |
     |                   or               |
     |           DX: Word Parameter       |
     |                   or               |
     |           DX: Address - Offset     |
     |           DS: Address - Segment    |
     +------------------------------------+

                  RETURN VALUES
                  =============

     +------------------------------------+
     | Register  AL: Byte Return          |
     |                   or               |
     |           AX: Word Return          |
     |                   or               |
     |           AX: Address - Offset     |
     |           ES: Address - Segment    |
     +------------------------------------+
     |           BX: Same as AX           |
     |           CX: Error Code           |
     +------------------------------------+
```

1.12 Error Handling

Most system functions return an Error Code to the calling process. Under MP/M-86, the CX register is reserved as the Error Code return register. Also under MP/M-86, there is one set of Error Codes common to all functions except those in the BDOS module. The BDOS functions have their own Error Codes which are explained in Section 2.15. The Error Codes for the non-BDOS MP/M-86 system functions are shown in Table 1-2.

Table 1-2. MP/M-86 Error Codes

CODE#	DEFINITION
0	NO ERROR
1	FUNCTION NOT IMPLEMENTED
2	ILLEGAL FUNCTION NUMBER
3	CAN´T FIND MEMORY
4	ILLEGAL SYSTEM FLAG NUMBER
5	FLAG OVERRUN
6	FLAG UNDERRUN
7	NO UNUSED QUEUE DESCRIPTORS LEFT IN QD TABLE
8	NO UNUSED QUEUE BUFFER AREA LEFT
9	CAN´T FIND QUEUE
10	QUEUE IN USE
11	QUEUE NOT ACTIVE
12	NO UNUSED PROCESS DESCRIPTORS LEFT IN PD TABLE
13	QUEUE ACCESS DENIED
14	EMPTY QUEUE
15	FULL QUEUE
16	CLI QUEUE MISSING
17	NO QUEUE BUFFER SPACE
18	NO UNUSED MEMORY DESCRIPTORS LEFT IN MD TABLE
19	ILLEGAL CONSOLE NUMBER
20	CAN´T FIND PD BY NAME
21	CONSOLE DOES NOT MATCH
22	NO CLI PROCESS
23	ILLEGAL DISK NUMBER
24	ILLEGAL FILE NAME
25	ILLEGAL FILE TYPE
26	CHARACTER NOT READY
27	ILLEGAL MEMORY DESCRIPTOR
28	BAD LOAD
29	BAD READ
30	BAD OPEN
31	NULL COMMAND
32	NOT OWNER
33	NO CODE SEGMENT IN LOAD FILE
34	ACTIVE PD
35	CAN´T TERMINATE
36	CAN´T ATTACH
37	ILLEGAL LIST DEVICE NUMBER
38	ILLEGAL PASSWORD

SECTION 2

THE MP/M-86 FILE SYSTEM

2.1 File System Overview

The Basic Disk Operating System (BDOS) supports a named file system on one to sixteen logical drives. Each logical drive is divided into two regions: a directory area and a data area. The directory area defines the files that exist on the drive and identifies the data area space that belongs to each file. The data area contains the file data defined by the directory. The directory area is subdivided into sixteen logically independent directories, which are identified by user numbers 0 through 15 respectively. In general, only files belonging to the current user number are "visible" in the directory. For example, the MP/M-86 DIR utility only displays files belonging to the current user number.

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

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

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

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

BDOS drive-related functions include those which select a drive as the default drive, compute a drive's free space, interrogate drive status and assign a Directory Label to a drive. The Directory Label for a drive controls whether file passwords are to be honored, and the type of date and time stamping to be performed for files on the drive. Also included in this category are functions to reset specified drives and to control whether other processes can reset particular drives. When a drive is reset, the next operation on the drive reactivates it by logging it in. The function of the log-in operation is to initialize the drive for file and directory operations. Under MP/M-86, a successful drive reset operation must be performed on drives that support removeable media before changing disks.

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

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

- Disk System Reset
- Drive Selection
- File Creation
- File Open
- File Close
- Directory Search
- File Delete
- File Rename
- Random or Sequential Read
- Random or Sequential Write
- Interrogate Selected Disks
- Set DMA Address
- Set/Reset File Indicators

Reset Drive
Access/Free Drive
Random Write With Zero Fill
Lock and Unlock Record
Set Multi-Sector Count
Set BDOS Error Mode
Get Disk Free Space
Chain To Program
Flush Buffers
Set Directory Label
Return Directory Label
Read and Write File XFCB
Set/Get Date and Time
Set Default Password
Return BDOS Serial Number

The following sections contain information on important topics related to the BDOS file system. The reader should be familiar with the content of these sections before attempting to use the system functions described individually in Section 6.

2.2 File Naming Conventions

Under MP/M-86, filenames consist of four parts: the drive select code (d), the filename field, the file type field, and the file password field. The general format for a command line file specification is shown below:

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

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

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

The PARSE FILENAME function recognizes certain ASCII characters as valid delimiters when it parses a file from a command line. The valid characters are shown in Table 2-1.

Table 2-1. Valid Filename Delimiters

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

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

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

It is not mandatory to follow the file naming conventions of MP/M-86 when creating or renaming a file with BDOS functions. However, the conventions must be used if the file is to be accessed from a command line. For example, the CLI function cannot locate a command file in the directory if its filename or type field contains a lower-case letter.

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

ASM	Assembler Source	LIB	Library File
BAK	ED Source Backup	LST	List File
BAS	Basic Source File	PLI	PL/I Source File
BRS	8080 Banked RSP File	PRL	Page Relocatable
CMD	8086 Command File	REL	Relocatable Module
COM	8080 Command File	RSP	Resident System Process
DAT	Data File	SPR	Syetem Page Relocatable
HEX	HEX Machine Code	SYM	SID Symbol File
H86	ASM-86 HEX File	SYS	System File
INT	Intermediate File	$$$	Temporary File

2.3 Disk Drive and File Organization

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

Table 2-2. Logical Drive Capacity

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

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

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

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

Source files under MP/M-86 are treated as a sequence of ASCII characters, where each "line" of the source file is followed by a carriage-return line-feed sequence (0DH followed by 0AH). Thus a single 128-byte record could contain several lines of source text. The end of an ASCII file is denoted by a ↑Z (1AH) or a real end-of-file, returned by the BDOS read operation. ↑Z characters embedded within machine code CMD files are ignored. The end of file condition returned by BDOS is used to terminate read operations.

2.4 File Control Block Definition

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

When making a file access or directory BDOS function call, a process passes an FCB address. The address is composed of two parts: register DX contains the offset, and DS contains the segment. The length of the FCB data area depends on the BDOS function. For most functions, the required length is 33 bytes. For random I/O functions and the COMPUTE FILE SIZE function, the FCB length must be 36 bytes. When either the BDOS OPEN or MAKE FILE functions specify a file is to be opened in Unlocked Mode, the FCB must be 35 bytes in length. The FCB format is shown below.

```
+-----------------------------------------------------------------+
|dr|f1|f2|...|f8|t1|t2|t3|ex|s1|s2|rc|d0|...|dn|cr|r0|r1|r2|
+-----------------------------------------------------------------+
 00 01 02 ... 08 09 10 11 12 13 14 15 16 ... 31 32 33 34 35
```

Figure 2-1. File Control Block Format

The fields in the FCB are defined as follows:

dr drive code (0 - 16).

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

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

t1,t2,t3 contain the file type in ASCII upper-case, with high bit = 0. t1', t2', and t3' denote the high bit of these positions, and are file attribute bits.

```
t1' = 1 => Read/Only file,
t2' = 1 => System file,
t3' = 1 => File has been archived.
```

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

cs contains the FCB checksum value for open FCBs.

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

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

d0...dn filled-in by MP/M-86, reserved for system use.

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

r0,r1,r2 optional Random Record Number in the range 0-242,143 (0 - 3FFFFH). ro,r1,r2 constitute a 18-bit value with low byte r0, middle byte r1, and high byte r2.

Note: The 2-byte File ID is returned in bytes r0 and r1 when a file is successfully opened in Unlocked Mode (see Section 2.9)

For BDOS directory functions, the calling process must initialize bytes 0 through 11 of the FCB before issuing the function call. The SET DIRECTORY LABEL and WRITE FILE XFCB functions also require the calling process to initialize byte 12. The BDOS RENAME FILE function requires the calling process to place the new file name and type in bytes 17 through 27.

BDOS OPEN or MAKE FILE function calls require the calling process to initialize bytes 0 through 12 of the FCB before issuing an OPEN FILE or MAKE FILE function call. Normally, byte 12 is set to zero. In addition, if the file is to be processed from the beginning using sequential read or write functions, byte 32 (cr) must be zeroed. After an FCB is activated by an open or make operation, the FCB should not be modified by the user. Open FCBs are checksum verified to protect the integrity of the file system. In general, if a process modifies an open FCB, the next read, write, or close function call will return with a checksum error (see Section 2.9 for more on FCB checksums). Normally, sequential read or write functions do not require initialization of an open FCB. However, random I/O functions require that a process set bytes 33 through 35 to the requested Random Record Number prior to making the function call.

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

As a general rule under MP/M-86, a process should close files as soon as they are no longer needed, even if they have not been modified. The BDOS file system maintains an entry in the system Lock List for each file opened by each process on the system. This entry is not removed from the system Lock List until the file is closed or the process owning the entry terminates. The BDOS file system uses this entry to prevent other processes from accessing the file unless the file was opened in a mode that supports shared access. Normally, a process must close a file before other processes on the system can access the file.

Keep in mind that the space in the system Lock List is a limited resource under MP/M-86. If a process attempts to open a file and no space exists in the system Lock List, or the process exceeds the process open file limit (specified during system generation), the BDOS denies the open operation and usually aborts the calling process.

The high-order bits of the FCB filename (f1',...,f8') and type (t1',t2',t3') fields are called attribute bits. Attributes bits are 1-bit boolean fields where 1 indicates on or true, and 0 indicates off or false. Attribute bits have two functions within the file system: as file attributes and interface attributes.

The file attributes (f1',...,f4' and t1',t2',t3') are used to indicate that a file has a defined attribute. These bits are recorded in a file's directory FCBs. File Attributes can only be set or reset by the BDOS SET FILE ATTRIBUTES function. When the BDOS MAKE FILE function creates a file, it initializes all file attributes to zero. A process can interrogate file attributes in an FCB activated by the BDOS OPEN FILE function or in directory FCBs returned by the BDOS SEARCH FOR FIRST and SEARCH FOR NEXT functions. Note: the BDOS file system ignores the file attribute bits when it attempts to locate a file in the directory.

The file attributes (t1',t2',t3') are defined by the file system as follows:

t1': Read/Only attribute

This attribute, if set, prevents write operations to a file.

t2': System Attribute

This attribute, if set, identifies the file as a MP/M-86 system file. System files are not normally displayed by the MP/M-86 DIR utility. In addition, user-zero system files can be accessed on a read/only basis from other user numbers (see Section 2.5).

t3': Archive Attribute

This attribute is designed for user-written archive programs. When a archive program copies a file to backup storage, it sets the archive attribute of the copied files. The file system automatically resets the archive attribute of a directory FCB that has been issued a write command. The archive program can test this attribute in each of the file's directory FCBs via the BDOS SEARCH FOR FIRST and SEARCH FOR NEXT functions. If all directory FCBs have the archive attribute set, it indicates that the file has not been modified since the previous archive. Note that the MP/M-86 PIP utility supports file archival.

Attributes f1' through f4' are available for definition by the user.

The interface attributes are f5' through f8'. These attributes cannot be used as file attributes. Interface attributes f5' and f6' are used to request options for BDOS calls requiring an FCB address in register DX. They are used by the BDOS OPEN, MAKE, CLOSE, and DELETE FILE functions. Table 2-3 shows the f5' and f6' interface attribute definitions for these functions.

Table 2-3. BDOS Interface Attributes

Function	Attribute
OPEN FILE	f5' = 1 : Open in Unlocked Mode f6' = 1 : Open in Read/only Mode
MAKE FILE	f5' = 1 : Open in Unlocked Mode f6' = 1 : Assign password to file
CLOSE FILE	f5' = 1 : Partial Close
DELETE FILE	f5' = 1 : Delete file XFCBs only

The interface attributes are discussed in detail for each of the above functions in Section 6. Attributes f5' and f6' are always reset when control is returned to the calling process. Interface attributes f7' and f8' are reserved for internal use by the BDOS file system.

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

2.5 User Number Conventions

The MP/M-86 User facility divides each drive directory into sixteen logically independent directories, designated as user 0 through user 15. Physically, all user directories share the directory area of a drive. In most other aspects, however, they are independent. For example, files with the same name can exist on different user numbers of the same drive with no conflict. However, a single file cannot reside under more than one user number.

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

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

User 0 has special properties under MP/M-86. With some restrictions, the file system automatically opens a file under user zero, if it is not present under the current user number. Of course, this action is only performed when the current user number is not zero. In addition, a file on user zero must have the system attribute (t2') set to be eligible for this operation. This procedure allows utilities that may include overlays and any other commonly accessed files to be placed on user zero, but be available for access from other user numbers. As a result, it eliminates the need for copying commonly needed utilities to all user numbers on a directory, and gives the MP/M-86 manager control over which user-zero files are directly accessible from other user numbers. Refer to Section 2.8 for more information on this topic.

2.6 Directory Labels and XFCBs

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

```
+-----------------------------------------------------+
|dr| file  | type  |pm|s1|s2|rc| password  | ts1 | ts2 |
|-----------------------------------------------------|
|00  01...   09..   12 13 14 15  16......   25.   29. |
+-----------------------------------------------------+
```

Figure 2-2. XFCB Format

The fields in the XFCB are defined as follows:

```
dr          - drive code (0 - 16)
file        - filename field
type        - file type field
pm          - password mode
              bit 7 - Read Mode
              bit 6 - Write Mode
              bit 5 - Delete Mode
              (bit references are right to left, relative to 0)
s1,s2,rc    - reserved for system use
password    - 8-byte password field (encrypted)
ts1         - 4-byte creation or access time stamp field
ts2         - 4-byte update time stamp field
```

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

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

dr	name	type	dl	s1	s2	rc	password	ts1	ts2
00	01..	09..	12	13	14	15	16......	25.	29.

Figure 2-3. Directory Label Format

```
dr          - drive code (0 - 16)
name        - Directory Label name
type        - Directory Label type
dl          - Directory Label data byte
              bit 7 - require passwords for files
              bit 6 - perform access time stamping
              bit 5 - perform update time stamping
              bit 4 - Make creates XFCBs
              bit 0 - Directory Label exists
              (bit references are right to left, relative to 0)
s1,s2,rc    - n/a
password    - 8-byte password field (encrypted)
ts1         - 4-byte creation time stamp field
ts2         - 4-byte update time stamp field
```

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

Note: The BDOS file system provides no function to read the Directory Label FCB directly. However, the Directory Label data byte can be read directly with the BDOS function, RETURN DIRECTORY LABEL. In addition, the BDOS search functions ('?' in FCB drive byte) can be used to find the Directory Label on the default drive. In the directory, the Directory Label is identified by a drive byte value (byte 0 in the FCB) equal to 32 (20H).

2.7 File Passwords

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

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

Table 2-4. Password Protection Modes

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

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

3 protection, and mode 2 protection implies mode 3 protection. All three modes require the user to specify the password to delete the file, rename the file, or to modify the file's attributes.

If the correct password is supplied, or if password protection is disabled by the Directory Label, then access to the BDOS functions is the same as for a file that is not password protected. In addition, the SEARCH FOR FIRST and SEARCH FOR NEXT functions are not affected by file passwords. Table 2-5 lists the BDOS functions that test for password.

Table 2-5. BDOS Functions That Test For Password

15.	OPEN FILE
19.	DELETE FILE
23.	RENAME FILE
30.	SET FILE ATTRIBUTES
100.	SET DIRECTORY LABEL
103.	WRITE FILE XFCB

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

2.8 File Date and Time Stamps

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

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

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

The BDOS file system uses the MP/M-86 internal date and time when it records a date and time stamp. The MP/M-86 TOD utility can be used to set the system date and time.

2.9 File Open Modes

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

Locked Mode:

A process can open a file in Locked Mode only if the file is not currently opened by another process. Once open in Locked Mode, no other process can open the file until it is closed. Thus, if a process successfully opens a file in Locked Mode, that process in effect owns the file until the file is closed or the process terminates. Files opened in Locked Mode support read and write operations unless the file is a read/only file (attribute t1' set) or the file is password protected in Write mode and the password is not supplied with the BDOS OPEN FILE call. In both of these cases, only read operations to the file are allowed. Note: Locked Mode is the default mode for opening files under MP/M-86.

Unlocked Mode:

A process can open a file in Unlocked Mode if the file is not currently open, or if the file has been opened by another process in Unlocked Mode. This mode allows more than one process to open the same file. Files opened in Unlocked Mode support read and write operations unless the file is a read/only file (attribute t1' set) or the file is password protected in Write mode and the password is not supplied with the BDOS OPEN FILE call. However, when a file opened in Unlocked Mode is extended by a write operation, the BDOS allocates space to the file in data block units, not in 128-byte record units as is normally the case. The BDOS record locking and unlocking functions are only supported for files opened in Unlocked Mode.

When opening a file in Unlocked Mode, a process must reserve 36 bytes in the FCB, because the OPEN FILE function returns a 2-byte value called the File ID in the r0 and r1 bytes of the FCB. The File ID is a required parameter for the BDOS record lock and record unlock commands.

Read/only Mode:

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

The OPEN FILE function performs the following steps for files opened in Locked or Read/only Mode. If the current user is non-zero, and the file to be opened does not exist under the current user number, the OPEN FILE function searches user zero for the file. If the file exist, under user zero and the file has the system attribute (t2') set, the file is opened under user zero. The open mode is automatically forced to Read/only when this is done. For more information on this, refer to Section 2.5.

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

Table 2-6 shows the definition of the FCB interface attributes f5' and f6' for the BDOS OPEN FILE function.

Table 2-6. FCB Interface Attributes F5' F6' OPEN FILE Function

f5' = 0,	f6' = 0	- open in Locked Mode (default mode)
f5' = 1,	f6' = 0	- open in Unlocked Mode
f5' = 0 or 1,	f6' = 1	- open in Read/only Mode

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

Table 2-7. **FCB Interface Attribute F6'**
MAKE FILE Function

```
+------------------------------------------------+
| f5' = 0 - open in Locked Mode (default mode)   |
| f5' = 1 - open in Unlocked Mode                |
+------------------------------------------------+
```

Note: the MAKE FILE function does not allow opening the file in Read/only Mode.

2.10 File Security

In general, the security measures implemented in the BDOS file system are intended to prevent accidental collisions between running processes. It is not possible to provide total security under MP/M-86 because the BDOS file system maintains file allocation information in open FCBs in the user's memory region, and MP/M-86 does not support memory protection. In the worst case, a program that "crashes" on MP/M-86 can take down the entire system. Therefore, MP/M-86 requires that all processes running on the system be "friendly." However, the BDOS file system is designed to ensure that multiple processes can share the same file system without interfering with each other. It does this in two ways:

- it performs checksum verification of open FCBs.
- it monitors all open files and locked records via the system Lock List.

User FCBs are checksum validated before I/O operations to protect the integrity of the file system from corrupted FCBs. The OPEN FILE and MAKE FILE functions compute and assign checksums to FCBs. The READ, WRITE, LOCK RECORD, UNLOCK RECORD and CLOSE FILE functions subsequently verify and recompute the checksums when the FCB changes. If the BDOS detects an FCB checksum error, it does not perform the requested command. Instead, it either terminates the calling process with an error, or if the process is in BDOS Return Error Mode (see Section 2.15), it returns to the process with an Error Code.

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

- prevent a process from deleting, renaming, or updating the attributes of another process's open file.

- prevent a process from opening a file currently opened by another process unless both processes open the file in Locked or Read/only Mode.
- prevent a process from resetting a drive on which another process has an open file.
- prevent a process from locking or updating a record currently locked by another process. Refer to Section 2.11 for more information on record locking and unlocking.

For reasons of efficiency, the file system verifies only for certain functions whether another process has the FCB specified file open. These functions are: OPEN FILE, MAKE FILE, DELETE FILE, RENAME FILE, and SET FILE ATTRIBUTES. For open FCBs, the FCB checksum controls whether the process can use the FCB. By definition, a valid FCB checksum implies that the file has been successfully opened and an entry for the file resides in the system Lock List. When a process closes a file permanently, the file system removes the file from the system Lock List and invalidates its FCB checksum field.

There are several other situations where the file system removes open file entries from the system Lock List for a process. For example, if a process makes a delete call for a file that it has open in Locked Mode, the file system deletes the file and also removes the file's entry from the system Lock List. Deleting an open file is not recommended practice under MP/M-86 but is supported for files opened in Locked Mode (the default open mode), to provide compatibility with software written under earlier releases of MP/M and CP/M. Note that the file system does not delete a file opened in Unlocked or Read/only Mode.

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

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

- A process deletes a file it has open in Locked Mode.
- A process renames a file it has open in Locked Mode.
- A process updates the attributes via the BDOS SET FILE ATTRIBUTES command of a file it has open in Locked Mode.
- A process issues a FREE DRIVE call for a drive on which it has an open file.

- A change in media is detected on a drive that has open files. This situation is a special case because a process cannot control whether it occurs and it can impact more than one process. Refer to Section 2.13 for more information on this situation.

The automatic verification of open FCBs by the file system after it purges a file entry from the system Lock List can affect performance. Each verification requires a directory search operation. Therefore, it is strongly recommended that these situations be avoided in new programs developed for MP/M-86.

2.11 Concurrent File Access

More than one process can access the same file if each process opens the file in the same shared access mode. BDOS supports two shared access modes, Unlocked and Read/only. Read/only Mode is functionally identical to the default Locked Mode except that more than one process can access the file and no process can change it. Files opened in Unlocked Mode present a more complex situation because a file opened in this mode can be modified by multiple processes concurrently. As a result, Unlocked Mode differs in some important ways from the other open modes.

When a process opens a file in Unlocked Mode, the file system returns a 2-byte field called the File ID in the r0 and r1 bytes of the FCB. The File ID is a required parameter of the BDOS LOCK RECORD and UNLOCK RECORD functions.

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

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

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

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

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

2.12 Multi-Sector I/O

The BDOS file system provides the capability to read or write multiple 128-byte records in a single BDOS function call. This multi-sector facility can be visualized as a BDOS "burst" mode, enabling a process to complete multiple I/O operations without interference from other running processes. The use of this facility in an application program can improve its performance, and also enhance overall system throughput. For example, the PIP utility performs its sequential I/O with a Multi-Sector Count of 8. Multi-sector I/O has its greatest impact, however, in the performance of sequential I/O processing on MP/M-86 systems that support record blocking/deblocking in their XIOS. Improved performance is achieved by eliminating the need for a large percentage of XIOS physical record pre-read operations.

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

The Multi-Sector Count determines the number of operations to be performed by the following BDOS functions:

- Sequential Read and Write functions
- Random Read and Write functions including WRITE WITH ZERO FILL and TEST AND WRITE RECORD
- LOCK RECORD and UNLOCK RECORD

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

2.13 XIOS Blocking and Deblocking

An optional physical record blocking and deblocking facility can be implemented as part of the XIOS when it is necessary to maintain physical records on disk in units greater than 128-bytes. In general, record blocking and deblocking in the XIOS is transparent to the BDOS file system as well as to programs that make BDOS file system calls.

If this facility is implemented, then the XIOS sends data to or receives data from the BDOS file system in logical 128-byte records, but accesses the disk with a larger physical record size. The XIOS uses an internal physical record buffer equal in size to the physical record size to buffer logical records. The process of building up physical records from 128-byte logical records is called blocking, and it is required for BDOS write operations. The reverse process is called deblocking and it is required for BDOS read operations. For BDOS write operations, the XIOS postpones the physical write operation for permanent drives (see Section 2.14) if the write operation is not to the directory. For BDOS read operations, the XIOS performs a physical read only if the current physical record buffer does not contain the requested logical record. In addition, if the physical record is "pending" as the result of a previous write operation, the XIOS performs a physical write operation prior to the read operation.

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

Note: The system automatically calls this function when a process terminates. In addition, the BDOS file system automatically makes a FLUSH BUFFERS call in the CLOSE FILE function.

2.14 Reset, Access and Free Drive

The BDOS functions DISK SYSTEM RESET, RESET DRIVE, ACCESS DRIVE, and FREE DRIVE allow a process to control when a drive's directory is to be reinitialized for file operations. When MP/M-86 is initiated by MPMLDR, all drives are initialized to the reset state. Subsequently, as drives are referenced, they are automatically logged-in by the file system. The log-in operation initializes the drive for BDOS file operations. In general, once a drive is logged-in, it is not necessary to relog the drive unless a disk media change is to be made. However, MP/M-86 requires that a successful drive reset be performed for a drive before a media change. If a drive is in the reset state when the media is changed, the next access to the drive logs in the drive. Note that the DISK SYSTEM RESET and RESET DRIVE functions have similar effects except that the DISK SYSTEM RESET function is directed to all drives on the system. The user can specify any combination of drives to be reset with the RESET DRIVE function.

Under MP/M-86, the drive reset operation is conditional in nature. Generally speaking, the file system cannot reset a drive for a process if another process has an open file on the drive. However, the exact action taken by a drive reset operation depends on whether the drive to be reset is permanent or removeable. MP/M-86 determines whether a drive is permanent or removeable by interrogating a bit in the drive's Disk Parameter Block (DPB) in the XIOS (refer to the MP/M-86 System's Guide for a detailed discussion of the DPB). A high-order bit of 1 in the DPB checksum vector size field designates the drive as permanent. Under MP/M-86, a drive's designation is critical to the reset operation, which is described below.

The BDOS first determines if there are any files currently open on the drive to be reset. If there are none, the reset takes place. Otherwise, if the drive is a permanent drive and if the drive is not read/only, the reset operation is not performed but a successful result is returned to the calling process. However, if the drive is removeable or read/only, the file system determines whether other processes have open files on the drive. If they do, the drive reset operation is denied and an Error Code is returned to the calling process. If all the files open on the drive belong to the calling process, the file system performs a "qualified" reset operation for the drive and returns a successful result to the calling process. This means that the next time the drive is accessed, the log-in operation is only performed if a media change is detected on the drive. The logic flow of the drive reset operation is shown in Figure 2-4.

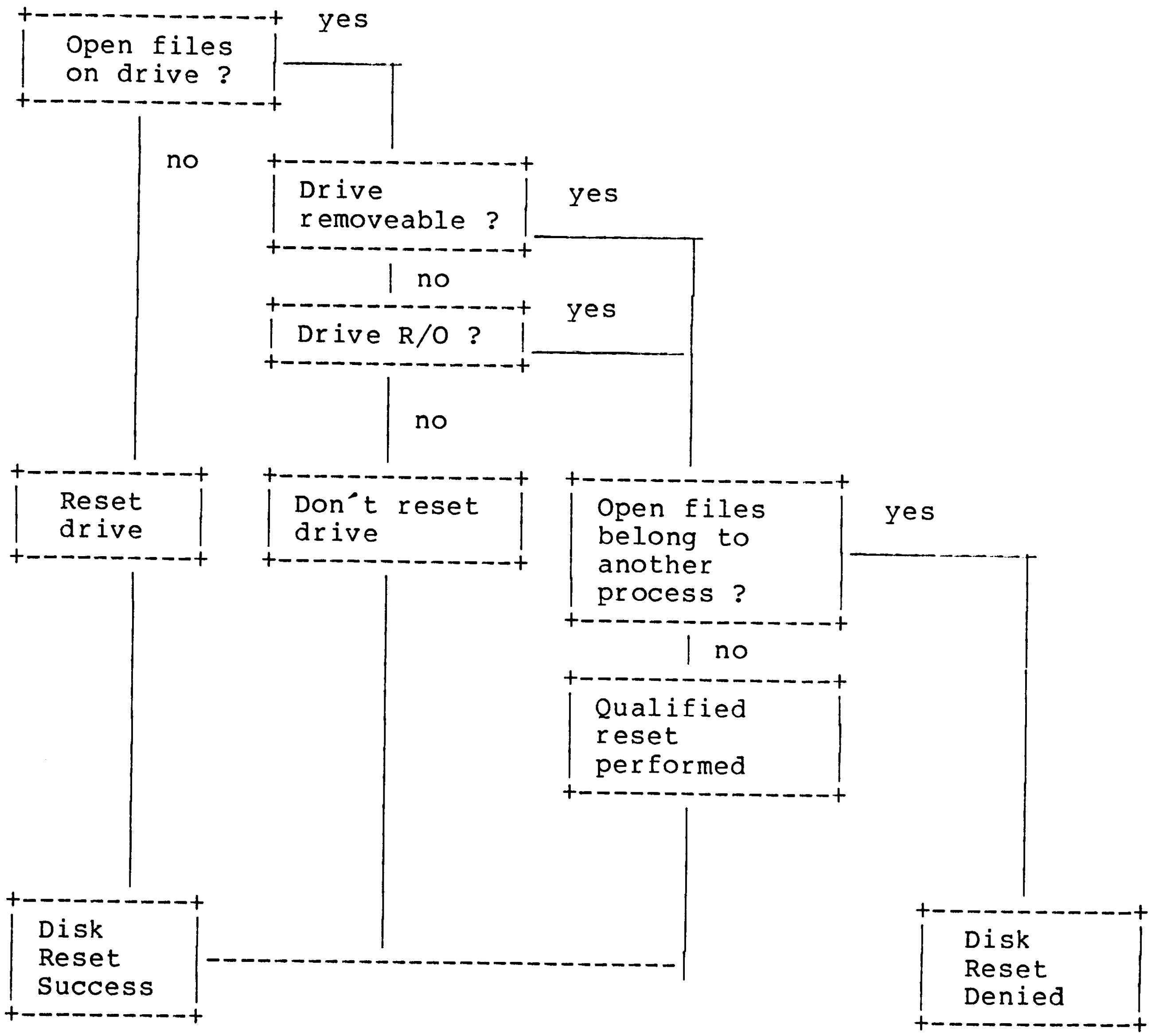

Figure 2-4. Disk System Reset

If the file system detects a media change on a drive after a qualified reset, it purges all open files on the drive from the system Lock List and subsequently verifies all open FCBs in file operations for the owning process (see Section 2.9). The drive is also relogged-in. In all other cases where a media change is detected on a drive, the file system performs the following steps: All open files on the drive are purged from the system Lock List, and all process owning a purged file are flagged for automatic open FCB verification. The drive is then placed in read/only status. It is not relogged-in until a drive reset is issued for the drive. Note: If a process references a file purged from the system Lock List in a BDOS command that requires an open FCB, it is returned an

FCB checksum error by the BDOS file system.

The ACCESS DRIVE and FREE DRIVE functions perform special actions under MP/M-86. The ACCESS DRIVE function inserts a "dummy" open file item into the system Lock List for each specified drive. While that item exists in the system Lock List, the drive cannot be reset by another process. The FREE DRIVE function purges the Lock List of all items including open file items belonging to the calling process on the specified drives. Any subsequent reference to those files by a BDOS function call requiring an open FCB results in a FCB checksum error return.

The WRITE PROTECT DISK function has special properties under MP/M-86. This function can be used to set the specified drive to read/only. However, MP/M-86 does not allow a process to set a drive read/only if another process has an open file on the drive. This applies to both removeable and permanent drives. If a process has successfully set a drive read/only, it can prevent other processes from resetting the drive by either opening a file on the drive or issuing an ACCESS DRIVE call for the drive. While the open file or "dummy" item belonging to the process resides in the system Lock List, no other process can reset the drive to take it out of read/only status.

2.15 BDOS Error Handling

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

1) It can return to the calling process with return codes in AX register identifying the error.

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

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

The file system handles the majority of errors it detects via method 1. The kinds of errors the file system handles via methods 2 and 3 are called "physical" and "extended" errors. The BDOS SET ERROR MODE function determines how the file system handles physical and extended errors. The BDOS Error Mode can exist in three states. In the default mode, the BDOS displays the error message and terminates the calling process (method 2). In Return Error Mode, the BDOS returns control to the calling process with the error identified in the AX register (method 1). In Return and Display Mode, the BDOS returns control to the calling process with the error identified in the AX register, and also displays the error message at the console (method 3). Both of the return modes ensure that MP/M-86 does not terminate the process because of a physical or extended error. The Return and Display Mode also allows the calling process to take advantage of the built-in error reporting of the BDOS file system. Physical and extended errors are displayed on the console in the following format:

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

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

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

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

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

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

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

The BDOS returns the "R/O" error whenever a process makes a write operation to a disk that is in read/only status. A drive can be placed in read/only status explicitly with the BDOS WRITE PROTECT DISK function, or implicitly if the file system detects a change in media on the drive.

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

- File Opened in Read/Only Mode
- File Currently Opened
- Close Checksum Error
- Password Error
- File Already Exists

- Illegal ? in FCB
- Open File Limit Exceeded
- No Room in System Lock List

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

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

The BDOS returns the "Close Checksum Error" message when the BDOS detects a checksum error in the FCB passed to the file system with a BDOS CLOSE FILE call.

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

The BDOS returns the "File Already Exists" error for the BDOS MAKE FILE and RENAME FILE functions when the BDOS detects a conflict on filename and type.

The BDOS returns the "Illegal ? in FCB" error whenever the BDOS detects a "?" in the filename or type field of the passed FCB for the BDOS RENAME FILE, SET FILE ATTRIBUTES, OPEN FILE and MAKE FILE functions.

The BDOS returns the "Open File Limit Exceeded" error when a process exceeds the file lock limit specified in the system Lock List during system generation. The OPEN FILE, MAKE FILE, and ACCESS DRIVE functions can return this error.

The BDOS returns the "No Room in System Lock List" error when no room for new entries exists within the system Lock List. The capacity of the system Lock List is a system generation parameter. The OPEN FILE, MAKE FILE, and ACCESS DRIVE functions can return this error.

The following paragraphs describe the error return code conventions of the BDOS file system functions. Most BDOS file system functions fall into three categories in regard to return codes; they return an Error Code, a Directory Code, or an Error Flag. The error conventions are designed to allow programs written for CP/M-86 to run without modification.

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

20. READ SEQUENTIAL
21. WRITE SEQUENTIAL
33. READ RANDOM
34. WRITE RANDOM
40. WRITE RANDOM WITH ZERO FILL
41. TEST AND WRITE RECORD
42. LOCK RECORD
43. UNLOCK RECORD

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

Table 2-8. BDOS Error Codes

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

* - returned only for files opened in Unlocked Mode
** - returned only by the LOCK RECORD function for files opened in Unlocked Mode

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

15. OPEN FILE
16. CLOSE FILE
17. SEARCH FOR FIRST
18. SEARCH FOR NEXT
19. DELETE FILE
22. MAKE FILE

23. RENAME FILE
30. SET FILE ATTRIBUTES
100. SET DIRECTORY LABEL
101. READ FILE XFCB
102. WRITE FILE XFCB

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

Table 2-9. BDOS Directory Codes

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

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

If the SET BDOS ERROR MODE function is used to place the BDOS in Return Error Mode, the following functions return an Error Flag in register AL on physical errors:

14. SELECT DISK
35. COMPUTE FILE SIZE
38. ACCESS DRIVE
46. GET DISK FREE SPACE
48. FLUSH BUFFERS
101. RETURN DIRECTORY LABEL DATA

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

Table 2-10. BDOS Error Flags

```
+-------------------------------------------+
|  00 : successful function                 |
| 255 : physical error : refer to register AH |
+-------------------------------------------+
```

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

```
+---------+
| N1 | N2 |
+---------+
```

Figure 2-5. Return Values - Register AH

where N1 denotes the high-order nibble and N2 denotes the low-order nibble. The following rules govern the assignment of values to N1 and N2.

N1 For functions that return Error Codes, the BDOS sets N1 to the number of sectors successfully read or written before the error is encountered. This information is returned only when a process uses the Set MULTI-SECTOR COUNT function to set the BDOS Multi-Sector Count to a value other than one; otherwise the BDOS sets N1 to zero. Successful read and write functions also set N1 to zero.

N1 Functions that return a Directory Code or an Error Flag set N1 to zero.

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

Table 2-11. BDOS Physical and Extended Errors

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

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

The following two functions represent a special case because they return an address in register AX.

27. GET ADDR(ALLOC)
31. GET ADDR(DISK PARMS)

When the BDOS is in Return Error Mode and it detects a physical error for these functions, it returns to the calling process with registers AX, and BX set to 255. Otherwise, they return no error code.

Under MP/M-86, the following functions also represent a special case.

13. RESET DISK SYSTEM
28. WRITE PROTECT DISK
37. RESET DRIVE

These functions return to the calling process with registers AL, and BL set to 255 if another process has an open file or has made a BDOS ACCESS DRIVE call that prevents the reset or write protect operation (see Section 2.14). If the BDOS is not in Return Error Mode, these functions also display an error message identifying the process that prevented the requested operation.

SECTION 3

TRANSIENT COMMANDS

3.1 Transient Process Load and Exit

A user can initiate a transient process by entering a command at a system console. The console's TMP then calls the CLI function, and passes to it the command line entered by the user. If the command is not resident, then the CLI function locates and then loads the proper CMD file (see the CLI function). The CLI function calls the PARSE FILENAME function which parses up to two filenames following the command and places the properly formatted FCBs at locations 005CH and 006CH in the Base Page of the initial Data Segment. The CLI function initializes memory, the Process Descriptor, and the User Data Area (UDA), and allocates a 96-byte stack area independent of the program, to contain the process's initial stack. MP/M-86 divides the DMA address into two parts: the DMA segment address, and the DMA offset. The CLI function initializes the default DMA base to the value of the initial Data Segment, and the default DMA offset to 0080H.

The CLI function creates the new process with a CREATE PROCESS call (Function 144), and sets the initial stack such that the process can execute a Far Return call to terminate. A process can also terminate by calling SYSTEM RESET (Function 0), or by calling TERMINATE (Function 143). A user may terminate a process by typing a single ↑C during line edited input. This has the same effect as the process calling Function 0.

3.2 Command File Format

A CMD file consists of a 128-byte Header Record followed immediately by the memory image. The command file Header Record is composed of 8 Group Descriptors (GDs), each 9 bytes long. Each Group Descriptor describes a portion of the program to be loaded. The format of the Header Record is shown in Figure 3-1.

```
+-------------------------------------------------------------------------+
| GD 1 | GD 2 | GD 3 | GD 4 | GD 5 | GD 6 | GD 7 | GD 8 | . . .  |
+-------------------------------------------------------------------------+

<------------------------ 128 Bytes ---------------------------->
```

Figure 3-1. CMD File Header Format

In Figure 3-1, GD 1 through GD 8 represent "Group Descriptors." Currently only the first 72 bytes of the Header Record are used. The remaining bytes are reserved for future facilities.

In Figure 3-1, each Group Descriptor corresponds to an independently loaded program unit and has the format shown in Figure 3-2.

```
   8-bit       16-bit      16-bit     16-bit    16-bit
+-------------------------------------------------------+
|  G-Form  |  G-Length  |  A-Base  |  G-Min  |  G-Max  |
+-------------------------------------------------------+
```

Figure 3-2. Group Descriptor Format

where G-Form describes the group format, or has the value zero if no more descriptors follow. If G-Form is non-zero, then the 8-bit value is parsed as two fields shown in Figure 3-3.

```
        G-Form:

+---------+---------+
|  4-bit  |  4-bit  |
+---------|---------+
| x x x x | G-Type  |
+---------+---------+
```

Figure 3-3. G-Form Format

The G-Type field determines the Group Descriptor type. The valid Group Descriptors have a G-Type in the range 1 through 9, as shown in Table 3-1.

Table 3-1. Group Descriptors

G-Type	Group Type
1	Code Group
2	Data Group
3	Extra Group
4	Stack Group
5	Auxiliary Group #1
6	Auxiliary Group #2
7	Auxiliary Group #3
8	Auxiliary Group #4
9	Shared Code Group
10	Unused, but Reserved
11	"
12	"
13	"
14	"
15	Escape Code for Additional Types

All remaining values in the Group Descriptor are given in increments of 16-byte paragraph units with an assumed low-order 0 nibble to complete the 20-bit address.

G-Length gives the number of paragraphs in the group. Given a G-length of 0080H, for example, the size of the group is 00800H = 2048D bytes.

A-Base defines the base paragraph address for a non-relocatable group.

G-Min/G-Max define the minimum and maximum size of the memory area to allocate to the group.

The memory model described by a Header Record is implicitly determined by the Group Descriptors (see Section 4.1). The 8080 Model is assumed when only a Code Group is present, since no independent Data Group is named. The Small Model is assumed when both a Code and Data Group are present, but no additional Group Descriptors occur. Otherwise, the Compact Model is assumed when the CMD file is loaded.

3.3 Base Page Initialization

The MP/M-86 Base Page contains default values and locations initialized by the CLI and PROGRAM LOAD functions, and used by the transient process.

The Base Page occupies the regions from offset 0000H through 00FFH relative to the initial Data Segment, and contains the values shown in Figure 3-4.

```
         L         M         H         L         H
   0         1         2         3         4         5         6
   +---------+---------+---------+---------+---------+---------+
0  |      Code   Length          |     Code    Base  |  M80    |
   +---------+---------+---------+---------+---------+---------+
6  |      Data   Length          |     Data base     |Reserved |
   +---------+---------+---------+---------+---------+---------+
C  |      Extra  Length          |     Extra base    |Reserved |
   +---------+---------+---------+---------+---------+---------+
12 |      Stack  Length          |     Stack Base    |Reserved |
   +---------+---------+---------+---------+---------+---------+
18 |           Aux 1             |     Aux 1         |Reserved |
   +---------+---------+---------+---------+---------+---------+
1E |           Aux 2             |     Aux 2         |Reserved |
   +---------+---------+---------+---------+---------+---------+
24 |           Aux 3             |     Aux 3         |Reserved |
   +---------+---------+---------+---------+---------+---------+
2A |           Aux 4             |     Aux 4         |Reserved |
   +---------+---------+---------+---------+---------+---------+
30 |     Bytes 30 through 4F are currently not used but are    |
   |     reserved for use by MP/M-86.                          |
   |                  .                                        |
   |                  .                                        |
   |                  .                                        |
   |                                                           |
   +---------+---------+---------+---------+---------+---------+
50 |  Drive  | Password 1  Addr  | P1 Len  | Password 2  Addr  |
   +---------+---------+---------+---------+---------+---------+
56 | P2 Len  |      Currently not used but reserved            |
   +---------+---------+---------+---------+---------+---------+
5C |                                                           |
   |                    Default FCB Area 1                     |
   |                         .                                 |
   |                         .                                 |
   +---------+---------+---------+---------+---------+---------+
6C |                                                           |
   |                    Default FCB Area 2                     |
   |                         .                                 |
   |                         .                                 |
   +---------+---------+---------+---------+---------+---------+
7C |   CR    | Random Record Number (opt)  | ///////////////////
   +---------+---------+---------+---------+---------+---------+
80 |                                                           |
   |                                                           |
   |              Default 128-byte DMA Buffer                  |
   |                                                           |
   |                                                           |
   +-----------------------------------------------------------+
```

Figure 3-4. MP/M-86 Base Page Values

The various fields within the Base Page are defined as follows:

- The M80 byte is a flag indicating whether the 8080 memory model was used during load. The values of the flag are defined as:

  1 = 8080 Model
  0 = not 8080 Model

  If the 8080 Model is used, the code length never exceeds 0FFFFH.

- The bytes marked Aux 1 through Aux 4 correspond to a set of four optional independent groups which may be required for programs which execute using the Compact Memory Model. The initial values for these descriptors are derived from the Header Record in the memory image file.

- Length is stored using the Intel convention (i.e. low, middle, and high bytes).

- Base refers to the address of the beginning of the segment.

- The Drive byte identifies the drive from which the transient program was read. 0 designates the default drive, while a value of 1 through 16 identifies drives A through P.

- Password 1 Addr (bytes 0051H-0052H) contains the address of the password field of the first command-tail operand in the default DMA buffer at 0080H. The CLI function sets this field to 0 if no password is specified.

- P1 Len (byte 0053H) contains the length of the password field for the first command-tail operand. The CLI function sets this to 0 if no password is specified.

- Password 2 Addr (bytes 0054H-0055H) contains the address of the password field of the second command-tail operand in the default DMA buffer at 0080H. The CLI function sets this field to 0 if no password is specified.

- P2 Len (byte 0056H) contains the length of the password field for the second command-tail operand. The CLI function sets this field to 0 if no password is specified.

- FCB Area 1 (bytes 005CH-007CH) is initialized by the CLI function for a transient program from the first command-tail operand of the command line (if it exists).

- FCB Area 2 (bytes 006CH-007CH) is initialized by the CLI function for a transient program from the second command-tail operand of the command line (if it exists). <u>Note:</u>

this area overlays the last 16 bytes of FCB Area 1. To use information in this area, the transient process must copy it to another location before using Area 1.

- The CR field (byte 007CH) contains the current record position used in sequential file operations with FCB area 1.

- The optional Random Record Number (bytes 007DH-007FH) is an extension of FCB Area 1 used in random record processing.

- The Default DMA buffer (bytes 0080H-00FFH) contains the command tail when the CLI function loads a transient program.

3.4 Parent/Child Reltionships

Under MP/M-86, when one process creates another process, there is a parent/child relationship between them. That is, the child process inherits all the default values of the parent process. This includes the default disk, user number, console, list device, and password. The child process will also inherit any Interrupt Vectors that the parent process has initialized.

SECTION 4

COMMAND FILE GENERATION

4.1 Transient Execution Models

The initial values of the segment registers are determined by which one of the three "memory models" is used by the transient process. The specific memory model is indicated in the CMD file Header Record. The three memory models are summarized in Table 4-1 below.

Table 4-1. MP/M-86 Memory Models

Model	Group Relationships
8080 Model	Code and Data Groups Overlap
Small Model	Independent Code and Data Groups
Compact Model	Three or More Independent Groups

The 8080 Model supports programs which are directly translated from an 8080 environment where code and data are intermixed. The 8080 Model consists of one group which contains all the code, data, and stack areas. Segment registers are initialized to the starting address of the region containing this group. The segment registers can, however, be managed by the application program during execution so that multiple segments within the Code Group can be addressed.

The Small Model is similar to that defined by Intel, where the program consists of an independent Code Group and a Data Group. The Code and Data Groups often consist of, but are not restricted to, single 64K-byte segments.

The Compact Model occurs when any of the Extra, Stack, or Auxiliary Groups are present in program. Each group may consist of one or more segments, but if any group exceeds one segment in size, or if Auxiliary Groups are present, then the application program must manage its own segment registers during execution in order to address all code and data areas.

The three models differ primarily in the manner in which the Operating System initializes the segment registers when it loads a transient process. The PROGRAM LOAD function determines the memory model used by a transient program by examining the program group usage, as described in the following sections.

4.1.1 The 8080 Memory Model

The 8080 Model is assumed when the transient program contains only a Code Group. In this case, the CLI function initializes the CS, DS, and ES registers to the beginning of the Code Group, and sets the SS and SP registers to a 96-byte initial stack area that it allocates. Note: the CLI function initializes the stack such that if the process executes a Far Return instruction, it will terminate. The CLI function sets the Instruction Pointer Register (IP) to 100H, thus allowing Base Page values at the beginning of the code group. Following program load, the 8080 Model appears as shown in Figure 4-1.

```
                 +------------------------+
     SS SP ----> |   96-BYTE STACK AREA   |
                 +------------------------+


                   +----------------+
                   |                |
                   |   CODE/DATA    |
                   |                |
                   |       .        |
                   |       .        |
                   |       .        |
                   |                |
                   |   CODE/DATA    |
                   |                |
           0100H   +----------------+
                   |  (IP = 0100H)  |
                   |                |
                   |   BASE PAGE    |
                   |                |
               0   +----------------+
                      CS   DS   ES
```

Figure 4-1. MP/M-86 8080 Memory Model

The intermixed code and data areas are indistinguishable. The Base Page values are described in Section 3-3. The following ASM-86 example shows how to code an 8080 Model transient program.

```
                cseg
                org     100h
                .
                .       (code)
        endcs   equ     $
                dseg
                org     offset endcs
                .
                .     (data)
                end
```

4.1.2 The Small Memory Model

The Small Model is assumed when the transient program contains both a Code and Data Group. (In ASM-86, all code is generated following a CSEG directive, while data is defined following a DSEG directive with the origin of the Data Segment independent of the Code Segment.) In this model, the CLI function sets the CS register to the beginning of the Code Group, the DS and ES registers to the beginning of the Data Group, and the SS and SP registers to a 96-byte initial stack area that it initializes. Following program load, the Small Model appears as shown in Figure 4-2.

```
                      +-------------------------+
  SS SP ----> |   96-BYTE STACK AREA    |
                      +-------------------------+


  +----------------+      +----------------+
  |                |      |                |
  |                |      |                |
  |                |      |       .        |
  |                |      |       .        |
  |                |      |       .        |
  |       .        |      |      DATA      |
  |       .        |      |                |
  |       .        |      |                |
  |                | 100H +----------------+
  |      CODE      |      |                |
  |                |      |   BASE PAGE    |
  |                |      |                |
  |  (IP = 0000H)  |      |                |
0 +----------------+    0 +----------------+
          CS                    DS   ES
```

Figure 4-2. MP/M-86 Small Memory Model

The machine code begins at CS+0000H, the Base Page values begin at DS+0000H, and the data area starts at DS+0100H. The following ASM-86 example shows how to code a Small Model transient program.

```
cseg
.
.      (code)
dseg
org      100h
.
.      (data)
end
```

4.1.3 The Compact Memory Model

The Compact Model is assumed when Code and Data Groups are present, along with one or more of the remaining Stack, Extra, or Auxiliary Groups. In this case, the CLI function sets the CS, DS, and ES registers to the base addresses of their respective areas, and the SS and SP registers to a 96-byte stack area it allocates. Figure 4-3 shows the initial configuration of the segments in the Compact Model. The values of the various segment registers can be programmatically changed during execution by loading from the initial values placed in Base Page, thus allowing access to the entire memory space.

```
                      +-------------------------+
          SS SP ----> |   96-BYTE STACK AREA    |
                      +-------------------------+

  +----------------+      +----------------+      +----------------+
  |                |      |                |      |                |
  |                |      |                |      |                |
  |                |      |                |      |                |
  |                |      |       .        |      |                |
  |                |      |       .        |      |                |
  |       .        |      |       .        |      |                |
  |       .        |      |                |      |       .        |
  |       .        |      |      DATA      |      |       .        |
  |                |      |                |      |       .        |
  |      CODE      |  100H+----------------+      |                |
  |                |      |                |      |      DATA      |
  |   (IP=0000H)   |      |   BASE PAGE    |      |                |
  |                |      |                |      |                |
0 +----------------+    0 +----------------+    0 +----------------+
          CS                      DS                      ES
```

Figure 4-3. MP/M-86 Compact Memory Model

If the transient program intends to use the Stack Group as a stack area, the SS and SP registers must be set upon entry. The SS and SP registers remain in the initial stack area, even if a Stack

Group is defined.

Although it may appear that the SS and SP registers should be set to address the Stack Group, there are two contradictions. First, the transient program may be using the stack group as a data area. In that case, the Far Call instruction used by the CLI function to transfer control to the transient program could overwrite data in the stack area. Second, the SS register would logically be set to the base of the group, while the SP would be set to the offset of the end of the group. However, if the Stack Group exceeds 64K the address range from the base to the end of the group exceeds a 16-bit offset value.

The following ASM-86 example shows how to code a Compact Model transient program.

```
cseg
.
.       (code)
dseg
org     100h
.
.       (data)
eseg
.
.       (more data)
sseg
.
.       (stack area)
end
```

4.2 GENCMD

The GENCMD utility creates a CMD file from an input HEX file. GENCMD is non-destructive. That is, it does not alter the original HEX file. The user invokes the GENCMD utility by typing

0A>**GENCMD filename {parameter-list}**

where the filename corresponds to the HEX input file with an assumed (and unspecified) file type of H86. GENCMD accepts optional parameters to specifically identify the 8080 Model and to describe memory requirements of each segment group. The GENCMD parameters are listed following the filename, as shown in the command line above where the parameter-list consists of a sequence of keywords and values separated by commas or blanks. The keywords are:

8080 CODE DATA EXTRA STACK X1 X2 X3 X4

The 8080 keyword forces a single Code Group so that the PROGRAM LOAD function sets up the 8080 Model for execution, thus allowing intermixed code and data within a single segment. The form of this command is

```
0A>GENCMD filename 8080
```

The remaining keywords follow the filename or the 8080 option and define specific memory requirements for each segment group, corresponding one-to-one with the segment groups defined in the previous section. In each case, the values corresponding to each group are enclosed in square brackets and separated by commas. Each value is a hexadecimal number representing a paragraph address or segment length in paragraph units denoted by hhhh, prefixed by a single letter which defines the meaning of each value:

Ahhhh	Load the group at absolute location hhhh
Bhhhh	The group starts at hhhh in the hex file
Mhhhh	The group requires a minimum of hhhh * 16 bytes
Xhhhh	The group can address a maximum of hhhh * 16 bytes

Generally, the CMD file Header Record values are derived directly from the HEX file and the parameters shown above need not be included. The following situations, however, require the use of GENCMD parameters.

- The 8080 keyword is included whenever ASM-86 is used in the conversion of 8080 programs to the 8086/8088 environment when code and data are intermixed within a single 64K segment, regardless of the use of CSEG and DSEG directives in the source program.

- An absolute address (A value) must be given for any group which must be located at an absolute location. Normally, this value is not specified since MP/M-86 cannot generally ensure that the required memory region is available, in which case the CMD file cannot be loaded.

- The B value is used when GENCMD processes a HEX file produced by Intel's OH86, or similar utility program that contains more than one group. The output from OH86 consists of a sequence of data records with no information to identify Code, Data, Extra, Stack, or Auxiliary groups. In this case, the B value marks the beginning address of the group named by the keyword, causing GENCMD to load data following this address to the named group (see the examples below). Thus, the B value is normally used to mark the boundary between Code and Data Segments when no segment information is included in the HEX file. Files produced by ASM-86 do not require the use of the B value since segment information is included in the HEX file.

- The minimum memory value (M value) is included only when the HEX records do not define the minimum memory requirements for the named group. Generally, the Code Group size is determined precisely by the data records loaded into the area. That is, the total space required for the group is defined by the range between the lowest and highest data byte addresses. The Data Group, however, may contain uninitialized storage at the end of the group and thus no data records are present in the HEX file which define the highest referenced data item. The highest address in the data group can be defined within the source program by including a "DB 0" as the last data item. Alternatively, the M value can be included to allocate the additional space at the end of the group. Similarly, the Stack, Extra, and Auxiliary Group sizes must be defined using the M value unless the highest addresses within the groups are implicitly defined by data records in the HEX file.

- The maximum memory size, given by the X value, is generally used when additional free memory may be needed for such purposes as I/O buffers or symbol tables. If the data area size is fixed, then the X parameter need not be included. In this case, the X value is assumed to be the same as the M value. The value XFFFF allocates the largest memory region available but, if used, the transient program must be aware that a three-byte length field is produced in the Base Page for this group where the high-order byte may be non-zero. Programs converted directly from an 8080 environment or programs that use a 2-byte pointer to address buffers should restrict this value to XFFF or less, producing a maximum allocation length of 0FFF0H bytes.

The following GENCMD command line transforms the file X.H86 into the file X.CMD with the proper Header Record:

0A>**gencmd x code[a40] data[m30,xfff]**

In this case, the Code Group is forced to paragraph address 40H, or equivalently, byte address 400H. The Data Group requires a minimum of 300H bytes, but can use up to 0FFF0H bytes, if available.

Assuming a file Y.H86 exists on drive B containing Intel HEX records with no interspersed segment information, the command

0A>**gencmd b:y data[b30,m20] extra[b50] stack[m40] x1[m40]**

produces the file Y.CMD on drive B by selecting records beginning at address 0000H for the Code Segment, with records starting at 300H allocated to the Data Segment. The Extra Segment is filled from records beginning at 500H, while the Stack and Auxiliary Segment #1 are uninitialized areas requiring a minimum of 400H bytes each. In this example, the data area requires a minimum of 200H bytes. Note again, that the B value need not be included if the Digital Research ASM-86 assembler is used.

4.3 Intel HEX File Format

GENCMD input is in Intel HEX format produced by both the Digital Research ASM-86 assembler and the standard Intel OH86 utility program (see Intel document #9800639-03 entitled "MCS-86 Software Development Utitities Operating Instructions for ISIS-II Users"). The CMD file produced by GENCMD contains a Header Record which defines the memory model and memory size requirements for loading and executing the CMD file.

An Intel HEX file consists of the traditional sequence of ASCII records in the following format:

```
+---+------+---------+-----+-----------------+-----+
| : | l l  | a a a a | t t | d d d   . . .  d | c c |
+---+------+---------+-----+-----------------+-----+
```

where the beginning of the record is marked by an ASCII colon, and each subsequent digit position contains an ASCII hexadecimal digit in the range 0-9 or A-F. The fields are defined in Table 4-1.

Table 4-1. Intel Hex Field Definitions

Field	Contents
ll	Record Length 00-FF (0-255 in decimal)
aaaa	Load Address
tt	Record Type: 00 data record, loaded starting at offset aaaa from current base paragraph 01 end of file, cc = FF 02 extended address, aaaa is paragraph base for subsequent data records 03 start address is aaaa (ignored, IP set according to memory model in use) The following are output from ASM-86 only: 81 same as 00, data belongs to Code Segment 82 same as 00, data belongs to Data Segment 83 same as 00, data belongs to Stack Segment 84 same as 00, data belongs to Extra Segment 85 paragraph address for absolute Code Segment 86 paragraph address for absolute Data Segment 87 paragraph address for absolute Stack Segment 88 paragraph address for absolute Extra Segment

Table 4-1. (continued)

Field	Contents
d	Data Byte
cc	Check Sum (00 - Sum of Previous Digits)

All characters preceding the colon for each record are ignored. (Additional HEX file format information is included in the ASM-86 User's Guide, and in Intel's document #9800821A entitled "MCS-86 Absolute Object File Formats.")

SECTION 5

RSP GENERATION

5.1 RSP Introduction

Resident System Processes are programs that can optionally become part of the MP/M-86 Operating System. They can be useful in several ways including creating a "turn key" system, autoloading programs when MP/M-86 is booted, creating customized user interfaces or "shells" at the consoles, monitoring hardware not supported in the XIOS, and avoiding disk loading time for often used commands.

The source code for the TMP (TERMINAL MESSAGE PROCESS) and ECHO RSPs is included in Appendices J and K, respectively. The reader should study these carefully while reading this section. The discussion of the CREATE PROCESS function (Function 144) in Section 6 is also helpful in understanding RSPs.

Resident System Processes are included with MP/M-86 during system generation. GENSYS searches the directory for all files with the file type .RSP and prompts the user to choose whether it will be included in the generated system file, MPM.SYS. An RSP file is created by generating a CMD file and renaming it. The GENSYS program is documented in the MP/M-86 System Guide.

5.2 RSP Memory Models

Under MP/M-86, there are two basic memory models for RSPs. They are similar to the 8080 and Small Models of transient programs. However, several important distinctions exist between the transient program and RSP memory models. The RSP has no equivalent to the Base Page of the transient program's Data Segment. The RSP is responsible for its own Process Descriptor (PD) and User Data Area (UDA). The system creates and initializes these data structures for the transient programs automatically at load time. RSPs, on the other hand, must have these structures initialized within their own Data Segments.

5.2.1 8080 Model RSP

The 8080 Model implies mixed code and data. When the system gives control of the CPU to an 8080 Model RSP, the Code, Data, Extra and Stack Segment registers are initialized to the same value. An 8080 Model RSP is generated by GENCMD with the 8080 option. GENSYS assumes the 8080 Model if the CMD file Header Record of the RSP has a single Code Group Descriptor and no other Group Descriptors (see Section 3.2). Throughout this section, when discussing an 8080 Model RSP, any reference to the Data Segment also refers to the Code Segment.

5.2.2 Small Model RSP

The Small Model RSP implies separate Code and Data Segments. When the system gives control of the CPU to a Small Model RSP, the Data, Extra and Stack Segment registers are initialized to the Data Segment while the Code Segment register is initialized to the Code Segment. There is no guarantee where GENSYS will place the Code Segment in memory relative to the Data Segment. The CMD Header Record for this kind of RSP must have both Data and Code Group Descriptors.

5.3 Multiple Copies of RSPs

At system generation, GENSYS can make up to 255 extra copies of an RSP such that each copy generates a separate process running under MP/M-86. GENSYS accomplishes this by making multiple copies of the RSP, and initializing each to be a separate RSP. The number of copies made by GENSYS can be fixed or dependent on a byte value in the System Data Area. To determine the number of copies to make, GENSYS looks at two fields in the RSP Header. The format of the RSP Header is shown in Figure 5-1.

```
byte:   0     1     2     3     4     5    ...    8
      +-----+-----+-----+-----+-----+----------------+
      |   LINK    |  SDATVAR  | NCP |    RESERVED    |
      +-----+-----+-----+-----+-----+----------------+
```

Figure 5-1. RSP Header Format

If the SDATVAR field is non-zero, it is used as an offset of a byte value in the System Data Area which contains the number number of copies to be generated. The offset should indicate a value that is set by the user during GENSYS. The TMP RSP uses this feature by placing the offset of the Number Of System Consoles field into the SDATVAR field. This way, a TMP is generated for each System Console specified by the user. If SDATVAR is 0 then the NCP byte in the RSP header is used as the number or extra copies to make. If both of these fields in the RSP Header are 0 then no extra copies are made and only a single RSP is created. The ECHO RSP is an example of the latter.

If the number of extra copies is determinted by GENSYS to be greater than 0, each copy of the RSP is given a unique copy number. The copy number is placed in the NCP field and the ASCII equivalent is appended to the end of the Process Descriptor NAME field of each copy. If there is not enough space for the number in the PD NAME, part of the PD NAME will be over written. For the example TMP RSP, GENSYS makes the specified number of copies and changes the NAME field in each copy to be "TMP0, TMP1, TMP2,...", and sets the NCP field to 0, 1, 2, ..., respectively.

5.3.1 8080 Model

When GENSYS makes copies of an 8080 Model RSP, the CS, DS, ES and SS fields in each copy's User Data Area are set to the paragraph address where the RSP will be in memory after loading.

5.3.2 Small Model

If multiple copies of a Small Model RSP are to be generated, GENSYS copies both the Code and Data Groups of the RSP, if the MEM field of the Process Descriptor is 0. See the CREATE PROCESS function for a description of the Process Descriptor format. GENSYS sets the UDA fields CS to the Code Segment of the RSP and DS, ES and SS to the Data Segment of the RSP.

5.3.3 Small Model with Shared Code

If a Small Model RSP has a non-zero MEM field in its Process Descriptor, the Code Segment is assumed to be reentrant. When copies are made of this type of RSP only the Data Group is copied. GENSYS sets the UDA CS field for each copy to the paragraph address of the one Code Segment for the RSP's. The DS, ES and SS, in each copied Data Segment, are set by GENSYS to the paragraph address of the Data Segment for that particular copy.

5.4 Creating and Initializing an RSP

An RSP that is to be invoked from a console, or through the CLI function (Function 150), must create a special queue called an RSP Command Queue. Such an RSP is called a Command RSP. This type of RSP usually performs some initialization routine and then goes into a loop. The initialization routine consists of creating and opening an RSP Command Queue as well as changing the priority to the default transient process priority. (Priority values with regard to RSPs are discussed below).

The first step of the loop is to read a message from the RSP Command Queue. The process that writes the message to the RSP Command Queue essentially activates the associated RSP. After the RSP returns from the READ QUEUE function call, it obtains the system resources it needs, such as the calling process's console. Typically, the RSP is assigned the console resource before a message is written to the RSP Command Queue. This is true however, only if the Process Descriptor name matches the queue name.

When the RSP completes its activities for the given command, it releases any system resources it has acquired, including the console, and re-starts the loop by reading from its RSP Command Queue. A Command RSP is a single process and is a serially re-usable resource; i.e., the RSP acts on one message at a time. When several processes attempt to invoke a single Command RSP, they will wait as described in the READ QUEUE and CONDITIONAL READ QUEUE function calls in Section 6. Note: it is certainly possible to create RSPs that are invoked differently and function differently than an Command RSP.

The format of the RSP Command Queue Message is shown in Figure 5-2.

```
byte:   0     1     2     3     4     . . .      130
      +-----+-----+-----+-----+---------------------+
      | PDADDRESS |     COMMAND TAIL   (129 bytes)   |
      +-----+-----+-----+-----+---------------------+
```

Figure 5-2. RSP Command Queue Message

The PDADDRESS is the offset relative to the System Data Area segment of the Process Descriptor of the process calling the RSP. A program that wants to invoke an RSP and is forming an RSP Command Queue Message, can find its Process Descriptor address by calling RETURN PD ADDRESS (Function 156). The COMMAND TAIL usually contains what the TMP sends to the CLI minus the command name, and is terminated with a zero byte.

When a command is entered at a console, the TMP performs a CLI function call. The CLI function attempts to open a Queue that has the RSP Flag on and has the same name as the command sent to the CLI. If the queue open is successful, the CLI function attempts to assign the calling process's console to a process with the same name as the command. If this step is also successful, the CLI function creates an RSP Command Queue Message with the command tail sent to the CLI from the TMP, and writes it to the RSP Command Queue (see the discussion of the CLI function in Section 6). A transient program can use a Command RSP in the same manner by writing directly to the appropriate RSP Command Queue. An advantage of using the CLI function is that it looks for an RSP first, and only searches on disk for a CMD file if the the RSP is not found.

When an RSP reads a RSP Command Queue Message, it will often need information about the calling process such as which console, list device, drive or user number to use. If an RSP is invoked through the CLI function, the RSP will have been assigned the calling process's console, but if the RSP Command Queue was written to directly, the calling process may or may not have assigned its console to the RSP. A Command RSP can use the PD address in the Command RSP Message to find out what the default devices of the calling process are. The RSP should release any resources it assigns to itself when it is finished.

The beginning of the RSP Data Segment has a fixed format starting at offset 0. This data structure is the RSP Header. Note that in the 8080 Model, the RSP Header is also in the Code Segment.

After the RSP Header is a Process Descriptor starting at offset 010H. A User Data Area and a stack must also be within the Data Segment, with the UDA placed at a paragraph boundary relative to the beginning of the Data Segment. If system functions assuming a default DMA buffer are used, a 128-byte DMA Buffer must also exist. The offset of this buffer is put in the DMA OFFSET field in the User Data Area. The DMA OFFSET can also be set by calling Function 26, SET DMA ADDRESS once the RSP is running. The DMA SEGMENT field in the UDA is set to the value in the DS field when a process is created. The beginning of the RSP Data Segment is shown in Figure 5-3.

```
       +-------------------------------------------+
  0H   |                RSP HEADER                 |
       +-------------------------------------------+
 10H   |            PROCESS DESCRIPTOR             |
       +-------------------------------------------+
 40H   |              USER DATA AREA               |
       +-------------------------------------------+
140H   |                  STACK                    |
       +-------------------------------------------+
```

Figure 5-3. Beginning of RSP Data Segment

The RSP Header must be located at offset zero in the RSP Data Segment, the RSP Process Descriptor must be at offset 010H. The RSP User Data Area must be on a even paragraph boundary.

5.4.1 The RSP Header

As discussed in Section 5.2, the number of copies made of an RSP is dependent on the values of the SDATVAR and NCP fields in the RSP Header. If no copies are desired, these fields must be zero. As a convienence, when MP/M-86 creates the RSP process, the LINK field in the RSP Header is set to the paragraph address of the System Data Area. The System Data Area can always be obtained by an RSP or transient program with the GET SYSTEM DATA ADDRESS function.

5.4.2 The RSP Process Descriptor

The RSP Process Descriptor should be initialed to zeros except for the PRIORITY, FLAGS, NAME, and UDA SEGMENT fields. The PRIORITY field is usually initialized to 190. This is higher than transient programs and TMPs, (200 and 198 respectively), but lower than the INIT process, which has one of the best possible priorities. The description of SET PRIORITY (Function 145) in Section 6 contains more information about system priority asssignments. Starting an RSP at a priority of 190 ensures that the RSP will be able to create and open an RSP Command Queue before it can be invoked through a TMP. RSPs such as ECHO, usually set their priority to 200 after creating and opening their RSP Command Queue and before attempting to read from the Queue. Note there are no guarantees about the order in which the RSP processes are created by the MP/M-86 Operating System. If one RSP must run before another, it must have a higher priority. Such is the case when one RSP uses a resource created by a second RSP; the second must run with a priority higher than the first.

The Process Descriptor SYS and KEEP Flags can be initialized in the RSP Data Segment. The SYS Flag allows a process to read and write to restricted system queues. This is discussed below with regard to RSP Command Queues. The KEEP flag signals to the Operating System that this process cannot be terminated. This flag is necessary if a RSP is not to be terminated when a ↑C is typed on a console being used by the RSP.

The NAME field of the RSPs Process Descriptor is 8 bytes long. It is assumed to be left justified and padded with blanks on the right. If an RSP Command Queue is going to be used to invoke the RSP through the CLI, the PD must have the same upper-case name as the Command Queue. The UDA field in the Process Descriptor must be the offset in paragraphs of the UDA relative to the RSP data segment.

5.4.3 The RSP User Data Area

The User Data Area must have the SP field set to the offset of a three-word "IRET structure", in the RSP's Data Segment. The offset is relative to the beginning of the Data Segment. The first of the three words is the offset of the code entry point for the RSP, relative to the beginning of the RSP Code Segment. MP/M-86 executes an IRET instruction to start the RSP using these three words for the IP, CS and Flag registers respectively. The CS value on the stack is initialized to be the CS field of the UDA while the Flag value is set to 0200H (interrupts on). The RSP stack must come immediately before these three words.

The initial values of the AX,BX,CX,DX,DI,SI and BP registers are taken from the appropriate fields in the UDA.

The DMA OFFSET field should be set to the offset of the DMA Buffer in the RSP's Data Segment. Except for the SP and DMA OFFSET fields, and possibly the AX,BX,CX,DX,DI,SI, and BP fields, the remainder of the UDA fields should be initialized to 0. The CS DS, ES and SS fields are set by GENSYS as discussed above.

5.4.4 The RSP Stack

The RSP must manage its own stack, which is assumed to lie within the RSP's Data Segment. This stack must be large enough to accommodate what the RSP code will need, plus four levels (eight bytes) to handle possible hardware interrupts. The three-word "IRET structure" pointed to by the SP field in the RSPs UDA, is considered part of the stack, since the 8086 Interrupt Return Instruction

(IRET) pops three words when the RSP starts execution.

5.4.5 The RSP Command Queue

The RSP's Command Queue contains information that determines when it will begin execution, and which console it will be attached to. If an RSP is to be accessable from a console via the TMP, the Command Queue name must be in upper-case. However the command tail put in an RSP Command Queue Message by the CLI, is not translated to upper-case. The FLAGS field in the RSP Command Queue Descriptor must have the RSP bit on. If this flag is not on, the CLI will not write a message to the RSP Command Queue, and will instead attempt to load a transient program. The KEEP flag should be set on to protect the RSP queue from inadvertent use of the DELETE QUEUE function.

The RESTRICTED flag makes a queue accessable only by privileged processes. Privileged processes have the SYS Flag on in their Process Descriptor. If the RESTRICTED Flag is on in an RSP Command Queue, then only privileged processes can invoke the related RSP. A lower-case letter in the RSP Command Queue name and the RESTRICTED Flag provide two methods of filtering access to an RSP.

The Queue Descriptor of the RSP Command Queue must have have a message length 131 bytes. The format of this message is shown above. The number of messages will usually be 1. If the Queue Descriptor is within 64K bytes of the beginning of the System Data Area, buffer space for the Queue Descriptor must be allocated in the RSP. The QBUFPTR field in the Queue Descriptor must be the offset of this buffer, relative to the beginning of the RSPs Data Segment. Also the Queue Buffer must be before the Queue Descriptor within the RSP Data Segment. The buffer size is the message length times the number of messages, usually 131 bytes.

An RSP can certainly create other queues besides the RSP Command Queue used with Command RSPs. However, any queue an RSP creates that lies within 64K of the System Data Area, must have a buffer area pointed to by the QBUFPTR field in its Queue Descriptor. To be safe, the buffer should come before the Queue Descriptor in the RSP's Data segment. It is assumed the QBUFPTR field points to a buffer that is also within 64K of the System Data Area. If the Queue Descriptor is farther than 64K from then System Data Area, MP/M-86 will use buffer space in the System Data Area. See the discussion of the MAKE QUEUE function call in Section 6 for more detail.

In order to open the RSP Command Queue and subsequently read from it, a Queue Parameter Block and its associated buffer must be allocated in the RSP's Data Segment. These structures are treated just as in a transient process. For any queues created by an RSP, it is stressed that the Queue Buffer areas associated with the Queue Descriptor and the Queue Parameter Block are separate, distinct areas of storage.

5.4.6 Multiple Processes within an RSP

An RSP can create child processes by calling CREATE PROCESS (function 144). Note that if the Process Descriptor of the process being created is within 64K bytes of the beginning of the System Data Area, the PD structure is used directly by MP/M-86. Otherwise the PD structure is copied into the PD table in the System Data Area.

5.5 Developing and Debugging an RSP

New RSPs should be debugged to as large extent as possible as transient, CMD type programs. The first RSP that the user attempts should be very simple, on the order of ECHO.

An RSP can be debugged in a similar manner as the XIOS, by running MP/M-86 under DDT86 which was loaded under a CP/M-86 system. Refer the MP/M-86 System Guide for more information about running MP/M-86 under CP/M-86. After reading the MPM.SYS file in under DDT86, the RSPSEG field of the System Data Area should be found. The paragraph address of the System Data Area is found in the ABS field of the Data Group Descriptor in the MPM.SYS command file Header. The CMD Header is described in Section 3.2 and the System Data Area is described in the MP/M-86 System Guide. The RSPSEG field contains the paragraph address of the Data Segment of the first RSP in a linked list of the RSPs included by GENSYS.

By using the Display Memory ("D") command of DDT86 to show memory at the segment RSPSEG, the name of the first RSP can be identified in the RSP's Process Descriptor. The LINK field in the RSP Header, which will be the first word in the RSPSEG segment, is the paragraph value of the next RSP's Data Segment. A zero in the LINK field means the end of the list of RSPs. Note that linkage information is lost once MP/M-86 is initialized. The LINK field of the RSP Header contains the System Data Segment once an RSP begins execution.

Once the RSP to be debugged is located, the initial code entry point may also be found. As discussed previously, the SP field in the RSP's UDA, is the offset from the beginning of the RSP's Data Segment, of the three-word "IRET structure". The first word of the "IRET structure" contains the initial value of the IP register when MP/M-86 creates the RSP process. The initial value of the CS register is in the CS field also in the RSP's UDA. Break points can now be set in the RSP, similar to break points set in XIOS functions.

SECTION 6

SYSTEM FUNCTION CALLS

This section contains a description of each of the MP/M-86 system functions, including the parameters a process must pass when calling the function, and the values the function returns to the process. The reader should be familiar with the material in Sections 1 through 5 before proceeding.

```
************************************************
*                                              *
*  FUNCTION 0:  SYSTEM RESET                   *
*                                              *
************************************************
*                                              *
*                   System Reset               *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 00H                       *
*                                              *
*  Return Values:                              *
*      Register  CX: Error Code                *
*                                              *
************************************************
```

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

The SYSTEM RESET function is implemented internally by calling the TERMINATE PROCESS function (Function 143) with the Termination Code set to zero.

Under CP/M-86, the SYSTEM RESET function has a further argument which allows a process not to release its memory. This is necessary to place a piece of code into memory that becomes an interface for later programs. This option is not included under MP/M-86. Memory segments are not recovered by the system until all processes that own the memory segment have released it.

```
**************************************************
*                                                *
*  FUNCTION 1:  CONSOLE INPUT                    *
*                                                *
**************************************************
*                                                *
*   Read a character from the default console    *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 01H                         *
*                                                *
*  Return Values:                                *
*      Register  AL: Character                   *
*                BL: Same as AL                  *
*                                                *
**************************************************
```

The CONSOLE INPUT function reads a character from the default console of the calling process. Before attempting the read, MP/M-86 internally calls the ATTACH CONSOLE function (Function 146) to verify ownership of the console. If the calling process does not own the console, it relinquishes the CPU resource until the attach operation is successful. Typically, a process that is created through the CLI function (Function 150) owns its default console when it begins execution.

MP/M-86 verifies ownership of the console resource in all console functions. This allows a user to type a ↑D character to detach a process. The detached process continues execution until it needs subsequent console I/O. It then waits until the console becomes available before continuing.

Function 1 echoes graphic characters read from the console. This includes the carriage return, line feed and backspace characters. It expands tab characters (↑I) in columns of eight characters, and checks for start/stop scroll (↑S/↑Q) and start/stop printer echo (↑P). It also checks for the terminate character (↑C) and the detach character (↑D). The terminate character causes the system to call the TERMINATE function with the termination code set to zero. Function 1 ignores the detach character if the calling process cannot be terminated (see Function 143). Function 1 does not return until a character is typed on the console. The system suspends the calling process until a character is ready.

```
*************************************************
*                                               *
*  FUNCTION 2:  CONSOLE OUTPUT                  *
*                                               *
*************************************************
*                                               *
*     Write a character to the default console  *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 02H                        *
*                DL: ASCII character            *
*                                               *
*************************************************
```

The CONSOLE OUTPUT function writes the specified character to the calling process' default console. As in the CONSOLE INPUT function (Function 1), MP/M-86 verifies that the calling process owns it default console before actually performing the operation. On output, Function 2 expands tabs in columns of eight characters and checks for start/stop scroll (↑S/↑Q) and start/stop printer echo (↑P). It also checks for the terminate character (↑C) and the detach character (↑D).

```
***************************************************
*                                                 *
*  FUNCTION 3:  RAW CONSOLE INPUT                 *
*                                                 *
***************************************************
*                                                 *
*   Read a character from the default console     *
*                  in Raw Mode                    *
*                                                 *
***************************************************
*                                                 *
*  Entry Parameters:                              *
*      Register  CL: 03H                          *
*                                                 *
*  Return Values:                                 *
*      Register  AL: Character                    *
*                BL: Same as AL                   *
*                                                 *
***************************************************
```

The RAW CONSOLE INPUT function reads a character from the default console of the calling process. As in the CONSOLE INPUT function (Function 1), MP/M-86 verifies ownership of the console before performing the operation. Calling Function 3 places the process in Raw Mode which means that no checking is done for special characters such as terminate or detach. Note: The process is taken out of Raw Mode as soon as a it calls a non-raw console function. Calling RAW CONSOLE INPUT will force the process to relinquish the CPU resource until a character is actually typed at the console.

MP/M-86 does not support the READER INPUT function because it treats all character I/O devices such as the Reader and the Punch as consoles. MP/M-86 places no practical limit to the number of Character I/O devices allowed to be configured with a system. (There is an absolute limit of 255 character I/O devices actually allowed).

```
*************************************************
*                                               *
*  FUNCTION 4:  RAW CONSOLE OUTPUT              *
*                                               *
*************************************************
*                                               *
*    Write a character to the default console   *
*                  in Raw Mode                  *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 04H                        *
*                DL: Character                  *
*                                               *
*************************************************
```

The RAW CONSOLE OUTPUT function writes a character to the default console of the calling process. MP/M-86 verifies ownership of the console before permitting the operation. Calling Function 4 places the process in Raw Mode which means that no checking is done for special characters such as terminate or detach.

MP/M-86 does not support the PUNCH OUTPUT function (see Function 3).

```
*************************************************
*                                               *
*  FUNCTION 5:  LIST OUTPUT                     *
*                                               *
*************************************************
*                                               *
*  Write a character to the default List device *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 05H                        *
*                DL: Character                  *
*                                               *
*************************************************
```

The LIST OUTPUT function writes the specified character to the default list device of the calling process. Before writing the character, the system internally calls ATTACH LIST, (Function 158) to verify that the calling process owns its default list device.

```
*************************************************
*                                               *
*  FUNCTION 6:  DIRECT CONSOLE I/O              *
*                                               *
*************************************************
*                                               *
*            Perform Direct Console I/O         *
*               with default console            *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 06H                        *
*                DL: 0FFH      (Input/          *
*                               Status)  or     *
*                    0FEH      (Status)  or     *
*                    0FDH      (Input)   or     *
*                    Character (Output)         *
*                                               *
*  Return Values:                               *
*      Register  AL: (Input/Status:)            *
*                         =   0H -No Character  *
*                         =   Character         *
*                    (Status:)                  *
*                         =   0H - No Character *
*                         = 0FFH - Ready        *
*                    (Input:)                   *
*                         =   Character         *
*                    (Output:)                  *
*                         No return value       *
*                BL: Same as AL                 *
*                                               *
*************************************************
```

The DIRECT CONSOLE I/O function allows the calling process to do Raw console I/O to its default console. MP/M-86 verifies that the calling process owns its default console before allowing any I/O.

A process calls the DIRECT CONSOLE I/O function by passing one of three different values shown below.

0FFH	console input command (If no character if ready, a 0H is returned).
0FEH	console status command (On return, register AL contains 00 if no character is ready; otherwise it contains FFH.)
0FDH	console input command (If no character is ready, the calling process waits until one is typed).
ASCII character	Function 6 assumes register DL contains a valid

ASCII character and sends it to the console.

There are two main differences between the DIRECT CONSOLE I/O function and the RAW CONSOLE functions (Function 3 and Function 4). First, CP/M-86 does not support the RAW CONSOLE functions but does support the DIRECT CONSOLE I/O function. Secondly, the DIRECT CONSOLE I/O does not allow totally transparent I/O because the calling process cannot output characters 0FFH, 0FEH or 0FDH. The RAW CONSOLE functions do allow totally transparent I/O when used in conjunction with the console status option in the DIRECT CONSOLE I/O function.

As with the RAW CONSOLE functions, the DIRECT CONSOLE I/O function places the calling process in Raw Mode, and special characters such as terminate and detach are not intercepted.

MP/M-86 performs a dispatch if the process sends a direct console input command (0FFH), and the function returns a 0 indicating that a character is not ready.

```
*************************************************
*                                               *
*  FUNCTION 7:  GET I/O BYTE                    *
*  FUNCTION 8:  SET I/O BYTE                    *
*                                               *
*************************************************
```

MP/M-86 does not support the GET I/O BYTE and SET I/O BYTE functions.

```
*************************************************
*                                               *
*  FUNCTION 9:  PRINT STRING                    *
*                                               *
*************************************************
*                                               *
*  Print an ASCII String to the default console *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 09H                        *
*                DX: STRING Address - Offset    *
*                DS: STRING Address - Segment   *
*                                               *
*************************************************
```

The PRINT STRING function prints an ASCII string starting at the indicated STRING address, and continuing until it reaches a dollar '$' character. Function 9 writes the string to the calling process's default console. MP/M-86 verifies that the calling process owns the console before writing the string. Function 9 recognizes any special characters such as terminate, detach or start/stop scroll. It also expands tabs in columns of eight characters as in the CONSOLE OUTPUT function (Function 2).

```
**************************************************
*                                                *
*  FUNCTION 10:  READ CONSOLE BUFFER             *
*                                                *
**************************************************
*                                                *
*  Read an edited line from the default console *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 0AH                         *
*                DX: BUFFER Address - Offset     *
*                DS: BUFFER Address - Segment    *
*                                                *
**************************************************
```

The READ CONSOLE BUFFER function reads characters from the calling process's default console and places them into the specified buffer. The format of the buffer is shown in Figure 6-1. Function 10 performs line editing functions on the line as it is read from the console. The READ CONSOLE BUFFER function completes a line and returns whenever it receives a terminator character from the console, or the maximum number of characters is reached. As in Function 1, the READ CONSOLE BUFFER function echoes all graphic characters read from the console. Note: MP/M-86 verifies that the calling process owns the default console before allowing I/O to begin.

```
      0     1                                  MAX + 2
+-----+-----+-----+-----+-----+-----+-/  /+-----+
| MAX |NCHAR|  CHARACTERS ...        \  \|     |
+-----+-----+-----+-----+-----+-----+-/  /+-----+
```

Figure 6-1. Console Buffer Format

MAX — Maximum number of characters that can be read into the buffer. This value must be initialized before calling the READ CONSOLE BUFFER function.

NCHAR — Actual number of characters read into the buffer as filled in by the READ CONSOLE BUFFER function.

CHARACTERS — Actual characters read from the console as filled in by the READ CONSOLE BUFFER function.

The READ CONSOLE BUFFER recognizes a number of special characters used in editing the input line as well as a set of special characters that actually control the calling process.

Line Editing Characters:

RUB/DEL	Removes the last character from the line and echoes it.
<↑E>	Echoes new line (a Carriage Return <↑M> and a Linefeed <↑J>) to the screen but does not affect the line buffer.
BACKSPACE <↑H>	Removes the last character from the line and backspaces over that character.
TAB <↑I>	Echoes enough spaces to place the next character position at a tab stop. Tab stops are fixed at every eighth character of the physical line.
LINE FEED <↑J>	Terminates the input line. The READ CONSOLE BUFFER function does not echo a terminating character nor does it place the character in the line buffer.
RETURN <↑M>	Terminates the input line.
REDRAW <↑R>	Retypes the current line after echoing a new line.
<↑U>	Removes all of the current line from the line buffer, echoes a new line, and starts all over again.
<↑X>	Removes all of the current line from the line buffer and echoes enough backspaces to return to the beginning of the line.

Process Control Characters:

TERMINATE (↑C)	Attempts to terminate the calling process with the TERMINATE function (FUNCTION 143). The Termination Code is set to zero. If the calling process does not terminate, the character is ignored. Function 10 only recognizes the detach character if it is the first character in the line.
DETACH (↑D)	Detaches the calling process from its default console. If there are any processes waiting to attach to the console, the process with the highest priority will then get the console. At this point, the system sends a message indicating which process now owns the console. The calling process can immediately recover the console only if no other processes are waiting. If the DETACH character is typed during the READ CONSOLE BUFFER function, the calling process effectively releases the CPU resource until the next detach character is typed. If the detach character is typed at other times, the process continues to execute in the background until console I/O is performed. At that time, the system internally calls ATTACH CONSOLE, and the process waits until a subsequent detach character allows the process to own the console again.

```
*************************************************
*                                               *
*  FUNCTION 11:  CONSOLE STATUS                 *
*                                               *
*************************************************
*                                               *
*    Obtain the status of the default console   *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 0CH                        *
*                                               *
*  Return Values:                               *
*      Register  AL: 01H character ready        *
*                    00H not ready              *
*                BL: Same as AL                 *
*                                               *
*************************************************
```

The CONSOLE STATUS function checks to see if a character has been typed at the default console of the calling process. If the calling process is not attached to its default console, the CONSOLE STATUS function will cause a dispatch to occur and return 00H (the not ready condition).

```
*************************************************
*                                               *
*  FUNCTION 12:  RETURN VERSION NUMBER          *
*                                               *
*************************************************
*                                               *
*         Return BDOS Version Number            *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 0CH                        *
*                                               *
*  Return Values:                               *
*      Register  AL: 30 (BDOS Version 3.0)      *
*                AH: 11 (MP/M-86)               *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The RETURN VERSION NUMBER function returns the BDOS file system version number, thereby allowing version independent programming.

The RETURN MPM VERSION function (Function 163) can be called to obtain the MP/M version number. Function 12 indicates the type of Operating System but not which version.

```
*************************************************
*                                               *
*  FUNCTION 13:  RESET DISK SYSTEM              *
*                                               *
*************************************************
*                                               *
*   Restore all File Systems to Reset State     *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 0DH                        *
*                                               *
*  Return Values:                               *
*      Register  AL: 0 if successful            *
*                    0ffH on error              *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The RESET DISK SYSTEM function restores the file system to a reset state where all the disk drives are set to read/write (see Functions 28 and 29), the default disk is set to drive A, and the default DMA address is reset to offset 080H relative to the current DMA segment address. This function can be used, for example, by an application program that requires disk changes during operation. RESET DRIVE (Function 37) can also be used for this purpose.

This function is conditional under MP/M-86. If another process has an open file on a removeable or read/only drive, the disk reset is denied and no drives are reset.

Upon return, if the reset operation is successful, the function returns a 0. Otherwise, it returns 0FFH (255 decimal). If the BDOS is not in the Return Error mode when an error occurs, (see Function 45), then the system displays an error message at the console, identifying the process owning an open file.

```
************************************************
*                                              *
*  FUNCTION 14:  SELECT DISK                   *
*                                              *
************************************************
*                                              *
*     Set calling process's default disk       *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 0EH                       *
*                DL: Selected Disk             *
*                                              *
*  Return Values:                              *
*      Register  AL: Error Flag                *
*                AH: Physical Error            *
*                BX: Same as AX                *
*                                              *
************************************************
```

The SELECT DISK function designates the specified disk drive as the default disk for subsequent BDOS file operations. The specified drive is set to 0 for drive A, 1 for drive B, and so-forth through 15 for drive P in a full 16-drive system. In addition, function 14 logs-in the designated drive if it is currently in the reset state. Logging-in a drive activates the drive's directory until the next RESET DISK SYSTEM or RESET DRIVE function call.

FCBs that specify drive code zero (dr = 00H) automatically reference the currently selected default drive. FCBs with drive code values between 1 and 16, however, ignore the selected default drive and directly reference drives A through P.

Upon return, register AL equal to 0 indicates the select operation was successful. If a physical error was encountered, the SELECT DISK function performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error mode is in the default mode, the system displays a message at the console identifying the error, and terminates the calling process. Otherwise, the SELECT DISK function returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical Error Codes:

01 : Permanent error
04 : Select error

```
*************************************************
*                                               *
*  FUNCTION 15:  OPEN FILE                      *
*                                               *
*************************************************
*                                               *
*               Open a Disk File                *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 0FH                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Directory Code             *
*                AH: Physical or Extended Error *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The OPEN FILE function activates the indicated FCB for a file that exists in the disk directory under the currently active user number, or user zero. The calling process passes the address of the FCB, with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the file name and type, and byte 12 specifying the extent. Normally, the process initilaizes byte 12 to zero. Interface attributes f5´ and f6´ of the FCB specify the mode in which the file is to be opened as shown below:

```
f5´ = 0,       f6´ = 0 - Open in Locked Mode (default mode)
f5´ = 1,       f6´ = 0 - Open in Unlocked Mode
f5´ = 0 or 1,  f6´ = 1 - Open in Read/only Mode
```

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

The OPEN FILE function performs the following steps for files opened in Locked or Read/only Mode. If the current user is non-zero, and the file to be opened does not exist under the current user number, the OPEN FILE function searches user zero for the file. If the file exists under user zero, and has the system attribute (t2´) set, the file is opened under user zero. The Open Mode is automatically set to Read/only when this is done.

The OPEN FILE function also performs the following action for files opened in Locked Mode when the current user number is zero. If the file exists in the directory under user zero, and has both the system attribute (t2´) set and the read/only attribute (t1´)

set, the Open Mode is automatically set to Read/only. Note that Read/only Mode implies the file can be concurrently accessed by other processes if they open the file in Read/only Mode.

If the open operation is successful, Function 15 activates the user´s FCB for read and write operations as follows: It copies the relevant directory information from the matching directory FCB into bytes d0 through dn of the FCB. It also computes a checksum and assigns it to the FCB. All BDOS functions that require an open FCB (e.g. READ SEQUENTIAL) verify that the FCB checksum is valid before performing their operation.

If the file is opened in Unlocked Mode, Function 15 sets bytes r0 and r1 of the FCB to a two-byte value called the File ID. The File ID is a required parameter for the BDOS LOCK RECORD and UNLOCK RECORD functions. If the Open Mode is forced to Read/only, Function 15 sets interface attribute f8´ to 1 in the user´s FCB. In addition, the function sets attribute f7´ to 1 if the referenced file is password protected in Write mode and the correct password was not passed in the DMA or did not match the default password. The BDOS does not support write operations for an activated FCB if interface attribute f7´ or f8´ is set to 1.

The BDOS file system also creates an open file item in the system Lock List to record a successful open file operation. While this item exists, no other process can delete, rename, or modify the file´s attributes. In addition, this item prevents other processes from opening the file if the file was opened in Locked Mode. It also requires that other processes match the file´s Open Mode if the file was opened in Unlocked or Read/only Mode. Normally, this item remains in the system Lock List until the file is permanently closed or the process that opened the file terminates.

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

Upon return, the OPEN FILE function returns a Directory Code in register AL with the value 0 through 3 if the open was successful, or 0FFH (255 decimal) if the file was not found. Register AH is set to 0 in both of these cases. If a physical or extended error was encountered, the OPEN FILE function performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is in the default mode, the system displays a message identifying the error at the console and the terminates the process. Otherwise, the OPEN FILE function returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended Error Codes:

01 : Permanent error
04 : Select error
05 : File is open by another process or by the current process in an incompatible mode

07 : File password error
09 : ? in the FCB file name or type field
10 : Process open file limit exceeded
11 : No room in the system Lock List

```
*************************************************
*                                               *
*  FUNCTION 16:  CLOSE FILE                     *
*                                               *
*************************************************
*                                               *
*              Close a Disk File                *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 10H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Directory Code             *
*                AH: Physical or Extended Error *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The CLOSE FILE function performs the inverse of the OPEN FILE function. The calling process passes the address of an FCB. The referenced FCB must have been previously activated by a successful OPEN or MAKE FILE function call (see Functions 15 and 22). Interface attribute f5' specifies how the file is to be closed as shown below:

f5' = 0 - Permanent close (default mode)
f5' = 1 - Partial close

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

If the CLOSE FILE function successfully performs the above steps, and if interface attribute f5' indicates that the close is permanent, it removes the file's item from the system Lock List. If the FCB was opened in Unlocked Mode, it also purges all record lock items belonging to the file from the system Lock List. By removing the file's Lock List item, the CLOSE FILE function invalidates the FCB's checksum to ensure the referenced FCB is not subsequently used with BDOS functions that require an open FCB (e.g. WRITE SEQUENTIAL).

The CLOSE FILE function makes an Update date and time stamp for the closed file under the following conditions: the referenced drive has a Directory Label that requests Update date and time stamping, the referenced file has an XFCB, the referenced drive is read/write, and a write operation to the file was made since the FCB was opened. None of these steps are performed for partial close operations (f5' = 1).

Upon return, the CLOSE FILE function returns a Directory Code in register AL with the value 0 to 3 if the close was successful, or 0FFH (255 Decimal) if the file was not found. Register AH is set to 0 in both of these cases. If a physical or extended error was encountered, the CLOSE FILE function performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is in the default mode, the system displays a message identifying the error at the console and terminates the calling process. Otherwise the CLOSE FILE function returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended Error Codes:

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

```
************************************************
*                                              *
*  FUNCTION 17:  SEARCH FOR FIRST              *
*                                              *
************************************************
*                                              *
*      Find the first file that matches        *
*             the specified FCB                *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 11H                       *
*                DX: FCB Address - Offset      *
*                DS: FCB Address - Segment     *
*                                              *
*  Return Values:                              *
*      Register  AL: Directory Code            *
*                AH: Physical or Extended Error *
*                BX: Same as AX                *
*                                              *
************************************************
```

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

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

If byte 0 of the referenced FCB is set to a question mark, Function 17 ignores the remainder of the referenced FCB and locates the first directory entry residing on the current default drive. All remaining directory entries can be located by making multiple SEARCH FOR NEXT calls. This type of search operation is not normally made by application programs, but it does provide complete flexibility to scan all current directory values. Note that this type of search operation must be performed to access a drive's

Directory Label (see Section 2.2.5).

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

If a physical error was encountered, the SEARCH FOR FIRST function performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is in the default mode, the system displays a message identifying the error at the console and terminates the calling process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical Error Codes:

01 : Permanent error
04 : Select error

```
*************************************************
*                                               *
*  FUNCTION 18:  SEARCH FOR NEXT                *
*                                               *
*************************************************
*                                               *
*    Find a subsequent file that matches the    *
* specified FCB of a previous Search for First  *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 12H                        *
*                                               *
*  Return Values:                               *
*      Register  AL: Directory Code             *
*                AH: Physical or Extended Error *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

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

```
*************************************************
*                                               *
*  FUNCTION 19:  DELETE FILE                    *
*                                               *
*************************************************
*                                               *
*              Delete a Disk File               *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 13H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Directory Code             *
*                AH: Physical or Extended Error *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The DELETE FILE function removes files and/or XFCBs that match the FCB addressed in register DX. The filename and type may contain ambiguous references (i.e., question marks in bytes f1 through t3), but the "dr" byte cannot be ambiguous, as it can in the SEARCH FOR FIRST and SEARCH FOR NEXT functions. Interface attribute f5´ specifies the type of delete operation to be performed as shown below:

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

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

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

A process can delete a file that it currently has open if the file was opened in Locked Mode. However, the BDOS returns a checksum error if the process makes a subsequent reference to the file with a BDOS function requiring an open FCB. Files open in Read/only or Unlocked Mode cannot be deleted by any process.

Upon return, the DELETE FILE function returns a Directory Code in register AL with the value 0 to 3 if the delete was successful, or 0FFH (255 Decimal) if no file matching the referenced FCB was found. Register AH is set to 0 in both of these cases. If a physical or extended error was encountered, Function 19 performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is the default mode, the system displays a message identifying the error at the console and terminates the calling process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended Error Codes:

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

```
*************************************************
*                                               *
*  FUNCTION 20:  READ SEQUENTIAL                *
*                                               *
*************************************************
*                                               *
*  Sequentially Read Records from a Disk File   *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*       Register  CL: 14H                       *
*                 DX: FCB Address - Offset      *
*                 DS: FCB Address - Segment     *
*                                               *
*  Return Values:                               *
*       Register  AL: Error Code                *
*                 AH: Physical Error            *
*                 BX: Same as AX                *
*                                               *
*************************************************
```

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

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

Upon return, the READ SEQUENTIAL function sets register AL to zero if the read operation was successful. Otherwise, register AL contains an error code identifying the error as shown below:

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

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

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

The function returns Error Code 10 if the referenced FCB failed the FCB checksum test.

The function returns Error Code 11 if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. The function only returns this error for files opened in Unlocked Mode.

The function returns Error Code 255 if a physical error was encountered and the BDOS is in Return Error mode or Return and Display Error mode (See Function 45). If the Error Mode is the default mode, the system displays a message at the console identifying the physical error, and terminates the calling process. When the function returns a physical error to the calling process, it is identified by the four low-order bits of register AH as shown below:

01 : Permanent error
04 : Select error

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

```
*************************************************
*                                               *
*  FUNCTION 21:  WRITE SEQUENTIAL               *
*                                               *
*************************************************
*                                               *
*   Sequentially Write Records to a Disk File   *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 15H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Error Code                 *
*                AH: Physical Error             *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The WRITE SEQUENTIAL function writes one to sixteen 128-byte data records beginning at the current DMA address into the file named by the specified FCB. The BDOS Multi-Sector Count (see Function 44) determines the number of 128-byte records that are written. The default is one record. The referenced FCB must have been previously activated by a BDOS OPEN or MAKE FILE function call.

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

Upon return, the WRITE SEQUENTIAL function sets register AL to zero if the write operation was successful. Otherwise, register AL contains an error code identifying the error as shown below:

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

The function returns Error Code 01 when it atempts to create a new extent that requires a new directory entry and no available directory entries exist on the selected disk drive.

The function returns Error Code 02 when it attempts to allocate a new data block to the file and no unallocated data blocks exist on the selected disk drive.

The function returns Error Code 08 if it attempts to write to a record locked by another process. The function only returns this error for files open in Unlocked Mode.

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

The function returns Error Code 10 if the referenced FCB failed the FCB checksum test.

The function returns Error Code 11 if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. The function only returns this error for files open in Unlocked Mode.

The function returns Error Code 255 if a physical error was encountered and the BDOS is in Return Error mode or Return and Display Error mode (See Function 45). If the Error Mode is the default mode, the system displays a message at the console identifying the physical error and terminates the calling process. When the function returns a physical error to the calling process, it is identified by the four low-order bits of register AH as shown below:

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

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

```
*************************************************
*                                               *
*  FUNCTION 22:  MAKE FILE                      *
*                                               *
*************************************************
*                                               *
*               Create a Disk File              *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 16H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Directory Code             *
*                AH: Physical or Extended Error *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

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

Interface attribute f5´ specifies the mode in which the file is to be opened. Interface attribute f6´ specifies whether a password is to be assigned to the created file. The interface attributes are summarized below:

f5´ = 0 - Open in Locked Mode (default mode)
f5´ = 1 - Open in Unlocked Mode
f6´ = 0 - Don´t assign password (default)
f6´ = 1 - Assign password to created file

When attribute f6´ is set to 1, the calling process must place the password in the first 8 bytes of the current DMA buffer and set byte 9 of the DMA buffer to the password mode (See Function 102).

The MAKE FILE function returns with an Error Code if the referenced FCB names a file that currently exists in the directory under the current user number. If there is any possibility of duplication, a DELETE FILE call should precede the MAKE FILE call.

If the make operation is successful, it activates the referenced FCB for file operations (opens the FCB) and initializes both the directory entry and the referenced FCB to an empty file. It also computes a checksum and assigns it to the FCB. BDOS functions that require an open FCB (e.g. WRITE RANDOM) verify that the FCB checksum is valid before performing their operation. If the file is opened in Unlocked Mode, the function sets bytes r0 and r1 in the FCB to a two-byte value called the File ID. The File ID is a required parameter for the BDOS LOCK RECORD and UNLOCK RECORD functions. Note that the MAKE FILE function intializes all file attributes to zero.

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

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

Upon return, the MAKE FILE function returns a Directory Code in register AL with the value 0 through 3 if the make operation was successful, or 0FFH (255 decimal) if no directory space was available. Register AH is set to zero in both of these cases. If a physical or extended error was encountered, the MAKE FILE function performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is the default mode, the system displays a message at the console identifying the error and terminates the calling process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended Error Codes:

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

```
*************************************************
*                                               *
*  FUNCTION 23:  RENAME FILE                    *
*                                               *
*************************************************
*                                               *
*              Rename a Disk File               *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 17H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Directory Code             *
*                AH: Physical or Extended Error *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

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

A process can rename a file that it has open if the file was opened in Locked Mode. However, the BDOS will return a checksum error if the process subsequently references the file with a function requiring an open FCB. A file open in Read/only or Unlocked Mode cannot be renamed by any process.

Upon return, the RENAME FILE function returns a Directory Code in register AL with the value 0 to 3 if the rename was successful, or 0FFH (255 Decimal) if the file named by the first file name in the FCB was not found. Register AH is set to zero in both of these cases. If a physical or extended error was encountered, the RENAME FILE function performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is the default mode, the system displays a message at the console identifying the error, and terminates the process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended Error Codes:

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

```
*************************************************
*                                               *
*  FUNCTION 24:  RETURN LOGIN VECTOR            *
*                                               *
*************************************************
*                                               *
*    Return Bit Map of Logged in Disk Drives    *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 18H                        *
*                                               *
*  Return Values:                               *
*      Register  AX: Login Vector               *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The RETURN LOGIN VECTOR function returns a bit map of currently logged-in disk drives. The login vector is a 16-bit value with the least significant bit corresponding to drive A, and the high-order bit corresponding to the 16th drive (drive P). A "0" bit indicates that the drive is not on-line, while a "1" bit indicates the drive is active. A drive is made active by either an explicit BDOS SELECT DISK call (Function 14), or an implicit selection when a BDOS file operation specifies a non-zero "dr" byte in the FCB.

```
*************************************************
*                                               *
*  FUNCTION 25:  RETURN CURRENT DISK            *
*                                               *
*************************************************
*                                               *
*  Return the Calling Process´s Default Disk    *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 19H                        *
*                                               *
*  Return Values:                               *
*      Register  AL: Login Vector               *
*                BL: Same as AL                 *
*                                               *
*************************************************
```

The RETURN CURRENT DISK function returns the calling process´s currently selected default disk. The disk numbers range from 0 through 15 corresponding to drives A through P.

```
************************************************
*                                              *
*  FUNCTION 26:  SET DMA OFFSET                *
*                                              *
************************************************
*                                              *
*     Set the Direct Memory Address Offset     *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 1AH                       *
*                DX: DMA Address - Offset      *
*                                              *
************************************************
```

"DMA" is an acronym for Direct Memory Address, which is often used in connection with disk controllers that directly access the memory of the computer to transfer data to and from the disk subsystem. Under MP/M-86, the current DMA is usually defined as the buffer in memory where a record resides before a disk write and after a disk read operation. If the BDOS Multi-Sector Count is equal to one (see Function 44), the size of the buffer is 128 bytes. However, if the BDOS Multi-Sector Count is greater than one, the size of the buffer must equal N * 128, where N equals the Multi-Sector Count.

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

When the CLI function initiates a transient program, it sets the DMA offset to 080H and the DMA Segment or Base to its initial Data Segment. RESET DISK SYSTEM (Function 13) also sets the DMA offset to 080H. The SET DMA OFFSET function can change this default value to another memory address. The DMA address remains at its current value until it is changed by a SET DMA OFFSET, Set DMA BASE or RESET DISK SYSTEM call.

```
*************************************************
*                                               *
*  FUNCTION 27:  GET ADDR (ALLOC)               *
*                                               *
*************************************************
*                                               *
*        Get Allocation Vector Address          *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 1BH                        *
*                                               *
*  Return Values:                               *
*      Register  AX: ALLOC Address - Offset     *
*                BX: Same as AX                 *
*                ES: ALLOC Address - Segment    *
*                                               *
*************************************************
```

MP/M-86 maintains an "allocation vector" in main memory for each active disk drive. Many programs commonly use the information provided by the allocation vector to determine the amount of free data space on a drive. Note, however, that the allocation information may be inaccurate if the drive has been marked read/only.

Function 27 returns the base address of the allocation vector for the currently selected drive. If a physical error is encountered when the BDOS Error Mode is one of the return modes (see Function 45), Function 27 returns the value 0FFFFH in AX.

GET DISK FREE SPACE (Function 46), can be used to directly return the number of free 128-byte records on a drive. In fact, the MP/M-86 utilities that display a drive's free space (STAT,SDIR, and SHOW) use Function 46 for that purpose.

```
*************************************************
*                                               *
*  FUNCTION 28:  WRITE PROTECT DISK             *
*                                               *
*************************************************
*                                               *
*       Set Default Disk to Read Only           *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 1CH                        *
*                                               *
*  Return Values:                               *
*      Register  AL: Return Code                *
*                BL: Same as AL                 *
*                                               *
*************************************************
```

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

The WRITE PROTECT DISK function is conditional under MP/M-86. If another process has an open file on the drive, the operation is denied and the function returns the value 0FFH to the calling process. Otherwise, it returns a 0. Note that a drive in the read/only state cannot be reset by a process if another process has an open file on the drive.

```
*************************************************
*                                               *
*  FUNCTION 29:  GET READ/ONLY VECTOR           *
*                                               *
*************************************************
*                                               *
*      Return Bit Map of Read Only Disks        *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 1DH                        *
*                                               *
*  Return Values:                               *
*      Register  AX: R/O Vector                 *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

Function 29 returns a bit vector indicating which drives have the temporary read/only bit set. The read/only bit is set either by a BDOS WRITE PROTECT DISK call, or by the automatic software mechanisms within MP/M-86 that detect changed disk media.

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

```
*************************************************
*                                               *
*  FUNCTION 30:  SET FILE ATTRIBUTES            *
*                                               *
*************************************************
*                                               *
*      Set the Attributes of a Disk File        *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 1EH                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Directory Code             *
*                BL: Same as AL                 *
*                                               *
*************************************************
```

The SET FILE ATTRIBUTES function is the only BDOS function that allows a program to manipulate file attributes. Other BDOS functions can interrogate these file attributes but cannot change them. The file attributes that can be set or reset by Function 30 are: f1' through f4', R/O (t1'), System (t2'), and Archive (t3'). The specified FCB contains a filename with the appropriate attributes set or reset. The calling process must ensure that it does not specify an ambiguous filename. In addition, if the specified file is password protected, the correct password must be placed in the first eight bytes of the current DMA buffer, or have been previously established as the default password (See Function 106).

Function 30 searches the FCB specified directory for an entry belonging to the current user number that matches the FCB specified name and type fields. The function then updates the directory to contain the selected indicators. File attributes t1', t2', and t3' are defined by MP/M-86. They are described in Section 2.2.4. Attributes f1' through f4' are not presently used, but may be useful for application programs, because they are not involved in the matching process used by the BDOS during OPEN FILE and CLOSE FILE operations. Indicators f5' through f8' are reserved for use as interface attributes.

This function is not performed if the file specified by the referenced FCB is currently open for another process. It is performed, however, if the referenced file is open for the calling process in Locked Mode. After successfully setting the attributes of a file opened by the calling process, the BDOS will return a checksum error on any subsequent file reference requiring an open FCB. Function 30 does not set the attributes of a file currently open in Read/only or Unlocked Mode for any process.

Upon return, Function 30 returns a Directory Code in register AL with the value 0 to 3 if the function was successful, or 0FFH (255 Decimal) if the file specified by the referenced FCB was not found. Register AH is set to zero in both of these cases. If a physical or extended error was encountered, the SET FILE ATTRIBUTES function performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is the default mode, the system displays a message at the console identifying the error and terminates the process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended Error Codes:

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

```
************************************************
*                                              *
*  FUNCTION 31:  GET ADDR (DISK PARMS)         *
*                                              *
************************************************
*                                              *
*    Return Address of Disk Parameter Block    *
*      for Calling Process's Default Disk      *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 1FH                       *
*                                              *
*  Return Values:                              *
*      Register  AX: DPB Address - Offset      *
*                    0FFFFH - on Physical Error *
*                BX: Same as AX                *
*                ES: DPB Address - Segment     *
*                                              *
************************************************
```

Function 31 returns the address of the XIOS-resident Disk Parameter Block (DPB) for the currently selected drive. (Refer to the MP/M-86 System Guide for the format of the DPB). The calling process can use this address to extract the disk parameter values for display or to compute the space on a drive.

If a physical error is encountered when the BDOS Error Mode is one of the return modes (See Function 45), Function 31 returns the value 0FFFFH.

```
*************************************************
*                                               *
*  FUNCTION 32:  SET/GET USER CODE              *
*                                               *
*************************************************
*                                               *
*      Set of Return the Calling Process's      *
*             Default User Code                 *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 20H                        *
*                DL: 0FFH to GET USER CODE      *
*                    User Code to SET           *
*                                               *
*  Return Values:                               *
*      Register  AL: Current User Code if SET   *
*                BL: Same as AL                 *
*                                               *
*************************************************
```

A process can change or interrogate the currently active user number by calling Function 32. If register DL = 0FFH, then the function returns the value of the current user number in register AL. The value can range of 0 to 15. If register DL is not 0FFH, then the function changes the current user number to the value of DL (modulo 16).

```
*************************************************
*                                               *
*  FUNCTION 33:  READ RANDOM                    *
*                                               *
*************************************************
*                                               *
*      Read Random Records from a Disk File     *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 21H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Error Code                 *
*                AH: Physical Error             *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The READ RANDOM function is similar to the READ SEQUENTIAL function except that the read operation takes place at a particular Random Record Number, selected by the 24-bit value constructed from the three-byte (r0, rl, r2) field beginning at postion 33 of the FCB. Note that the sequence of 24 bits is stored with the least significant byte first (r0), the middle byte next (rl), and the high byte last (r2). The Random Record Number can range from 0 to 242,143. This corresponds to a maximum value of 3 in byte r2.

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

If the BDOS Multi-Sector count is greater than one (See Function 44), the READ RANDOM function reads multiple consecutive records into memory beginning at the current DMA. Function 33 automatically increments the r0,rl, and r2 field of the FCB to read each record. However, it restores the FCB´s Random Record Number to the first record´s value upon return to the calling process. Upon return, the READ RANDOM function sets register AL to zero if the read operation was successful. Otherwise, register AL contains one of the following error codes:

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

The function returns Error Code 01 when the it accesses a data block that has not been previously written.

The function returns Error Code 03 when it cannot close the current extent prior to moving to a new extent.

The function returns Error Code 04 when a read random operation accesses an extent that has not been created.

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

The function returns Error Code 10 if the referenced FCB failed the FCB checksum test.

The function returns Error Code 11 if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. The function only returns this error for files open in Unlocked Mode.

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

01 : Permanent Error
04 : Select Error

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

```
*************************************************
*                                               *
*  FUNCTION 34:  WRITE RANDOM                   *
*                                               *
*************************************************
*                                               *
*     Write Random Records from a Disk File     *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 22H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Error Code                 *
*                AH: Physical Error             *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

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

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

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

If the BDOS Multi-Sector count is greater than one (see Function 44), the WRITE RANDOM function reads multiple consecutive records into memory beginning at the current DMA. The function automatically increments the r0,r1, and r2 field of the FCB to write each record. However, it restores the FCB's Random Record Number to the first record's value upon return to the calling process. Upon return, the WRITE RANDOM function sets register AL to zero if the write operation was successful.

Otherwise, register AL contains one of the following Error Codes:

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

The function returns Error Code 02 when it attempts to allocate a new data block to the file and no unallocated data blocks exist on the selected disk drive.

The function returns Error Code 03 when it cannot close the current extent prior to moving to a new extent.

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

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

The function returns Error Code 08 when it attempts to write to a record locked by another process. The function only returns this error is only returned for files open in Unlocked Mode.

The function returns Error Code 10 if the referenced FCB failed the FCB checksum test.

The function returns Error Code 11 if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. The function only returns this error for files open in Unlocked Mode.

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

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

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

```
*************************************************
*                                               *
*  FUNCTION 35:  COMPUTE FILE SIZE              *
*                                               *
*************************************************
*                                               *
*       Compute the size of a Disk File         *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 23H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Error Flag                 *
*                AH: Physical or Extended Error *
*                BX: Same as AX                 *
*                Random Record Field of FCB Set *
*                                               *
*************************************************
```

The COMPUTE FILE SIZE function determines the "virtual" file size, which is, in effect, the address of the record immediately following the end of the file. The "virtual" size of a file corresponds to the physical size if the file is written sequentially. If the file is written in random mode, gaps may exist in the allocation, and the file may contain fewer records than the indicated size. For example, if a single record with record number 262,143 (the MP/M-86 maximum) is written to a file using the WRITE RANDOM function, then the "virtual" size of the file is 262,144 records even though only 1 data block is actually allocated.

To compute file size, the calling process passes the address of a FCB in random mode format (bytes r0, r1 and r2 present). Note that the FCB must contain an unambiguous filename and type. Function 35 sets the random record field of the FCB to the Random Record Number + 1 of the last record in the file. If the r2 byte is set to 04, then the file contains the maximum record count 262,144.

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

<u>Note:</u> the BDOS does not require the file to be open in order to use Function 35.

Upon return, Function 35 returns a zero in register AL if the file specified by the referenced FCB was found, or a 0FFH in register AL if the file was not found. Register AH is set to zero in both of these cases. If a physical or extended error was encountered, Function 35 performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is the default mode, the system displays a message at the console identifying the error and terminates the process. Otherwise, Function 35 returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended Errors Codes:

01 : Permanent error
04 : Select error
09 : ? in filename or type field

```
**************************************************
*                                                *
*  FUNCTION 36:  SET RANDOM RECORD               *
*                                                *
**************************************************
*                                                *
*     Return the Random Record Number of the     *
*     Next Record to Access in a Disk File       *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 24H                         *
*                DX: FCB Address - Offset        *
*                DS: FCB Address - Segment       *
*                                                *
*  Return Values:                                *
*                Random Record Field of FCB Set  *
*                                                *
**************************************************
```

The SET RANDOM RECORD function returns the Random Record Number of the next record to be accessed from a file that has been read or written sequentially to a particular point. The function returns this value in the random record field (bytes r0, r1, and r2) of the addressed FCB. Function 36 can be useful in two ways.

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

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

```
*************************************************
*                                               *
*  FUNCTION 37:  RESET DRIVE                    *
*                                               *
*************************************************
*                                               *
*         Reset Specified Disk Drives           *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 25H                        *
*                DX: Drive Vector               *
*                                               *
*  Return Values:                               *
*                AL: Return Code                *
*                BL: Same as AL                 *
*                                               *
*************************************************
```

The RESET DRIVE function is used to programmatically restore specified drives to the reset state (a reset drive is not logged-in and is in read/write status). The passed parameter in register DX is a 16-bit vector of drives to be reset, where the least significant bit corresponds to the first drive A, and the high-order bit corresponds to the sixteenth drive, labelled P. Bit values of "1" indicate that the specified drive is to be reset.

This function is conditional under MP/M-86. If another process has a file open on a drive to be reset, and the drive is removeable or read/only, the DRIVE RESET function is denied and no drives are reset.

Upon return, if the reset operation is successful, Function 37 sets register AL to 0. Otherwise, it sets register AL to 0FFH (255 decimal). If the BDOS is not in Return Error mode (see Function 45), then the system displays an error message at the console identifying the process owning an open file.

```
**************************************************
*                                                *
*  FUNCTION 38:  ACCESS DRIVE                    *
*                                                *
**************************************************
*                                                *
*          Access Specified Disk Drives          *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*       Register  CL: 26H                        *
*                 DX: Drive Vector               *
*                                                *
*  Return Values:                                *
*                 AL: Return Code                *
*                 AH: Extended Error             *
*                 BL: Same as AL                 *
*                                                *
**************************************************
```

The ACCESS DRIVE function inserts a special open file item into the system Lock List for each specified drive. While the item exists in the Lock List, the drive cannot be reset by another process. As in Function 37, the calling process passes the drive vector in register DX. The format of the drive vector is the same as that used in Function 37.

The ACCESS DRIVE function inserts no items if insufficient free space exists in the Lock List to support all the new items or if the number of items to be inserted puts the calling process over the Lock List open file maximum. This maximum is a MP/M-86 GENSYS option. If the BDOS Error Mode is the default mode (see Function 45), the system displays a message at the console identifying the error and terminates the calling process. Otherwise, the ACCESS DRIVE function returns to the calling process with register AL set to 0FFH and register AH set to one of the following values.

```
10 : Process Open File limit exceeded
11 : No room in the system Lock List
```

If the ACCESS DRIVE function is successful, it sets register AL to 0.

```
************************************************
*                                              *
*  FUNCTION 39:  FREE DRIVE                    *
*                                              *
************************************************
*                                              *
*          Free Specified Disk Drives          *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 27H                       *
*                DX: Drive Vector              *
*                                              *
************************************************
```

The FREE DRIVE function purges the System Lock List of all file and locked record items that belong to the calling process on the specified drives. As in Function 38, the calling process passes the drive vector in register DX.

Function 39 does not close files associated with purged open file Lock List items. In addition, if a process references a "purged" file with a BDOS function requiring an open FCB, the function returns a checksum error. A file that has been written to should be closed before making a FREE DRIVE call to the file's drive. Otherwise data may be lost.

```
*************************************************
*                                               *
*  FUNCTION 40:  WRITE RANDOM WITH ZERO FILL    *
*                                               *
*************************************************
*                                               *
*     Write a Random Record to a Disk File      *
*    and Pre-Fill New Data Blocks With Zeros    *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 28H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Error Code                 *
*                AH: Physical Error             *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The WRITE RANDOM WITH ZERO FILL function is similar to the WRITE RANDOM function (Function 34) with the exception that it fills a previously unallocated data block with zeros before writing the record. If this function has been used to create a file, records accessed by a READ RANDOM function that contain all zeros identify unwritten Random Record Numbers. Unwritten random records in allocated data blocks of files created using the WRITE RANDOM function contain uninitialized data.

```
*************************************************
*                                               *
*  FUNCTION 41:  TEST AND WRITE RECORD          *
*                                               *
*************************************************
*                                               *
* Verify Contents of Current Record Before Write*
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 29H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Error Code                 *
*                AH: Physical Error             *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The TEST AND WRITE RECORD function provides a means of verifying the current contents of a record on disk before updating it. The calling process must set bytes r0, r1, and r2 of the FCB addressed by register DX to the Random Record Number of the record to be tested. The original version of the record (i.e. the record to be tested) must reside at the current DMA address, followed immediately by the new version of the record. The record size can range from 128 bytes to sixteen times that value depending on the BDOS Multi-Sector Count (see Function 44).

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

The TEST AND WRITE RECORD function is intended for use in situations where more than one process has read/write access to a common file. This situation is supported under MP/M-86, when more than one process opens the same file in unlocked mode. Function 41 is a logical replacement for the record lock/unlock sequence of operations because it prevents two processes from simultaneously updating the same record. Note that this function is also supported for files open in Locked Mode to provide compatibility between MP/M-86 and CP/M-86.

Upon return, the TEST AND WRITE RECORD function sets register AL to zero if the function was successful.

Otherwise, register AL contains one of the following Error Codes:

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

The function returns Error Code 01 when it accesses a data block which has not been previously written.

The function returns Error Code 03 when it cannot close the current extent prior to moving to a new extent.

The function returns Error Code 04 when a read operation accesses an extent that has not been created.

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

The function returns Error Code 07 when the record to be updated does not match the record on disk.

The function returns Error Code 08 if the specified record is locked by another process. The function only returns this error for files opened in Unlocked Mode.

The function returns Error Code 10 if the referenced FCB failed the FCB checksum test.

The function returns Error Code 11 if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. The function only returns this error for files opened in Unlocked Mode.

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

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

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

```
*************************************************
*                                               *
*  FUNCTION 42:  LOCK RECORD                    *
*                                               *
*************************************************
*                                               *
*        Lock Records in a Disk File            *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 2AH                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Error Code                 *
*                AH: Physical Error             *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The LOCK RECORD function locks one or more consecutive records so that no other program with access to the records can simultaneously lock or update them. This function is only supported for files open in Unlocked Mode. If it is called for a file open in Locked or Read/only Mode, no locking action is performed and a successful result is returned. This is done to provide compatibility between MP/M-86 and CP/M-86.

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

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

A MP/M-86 system generation parameter specifies the maximum number of records that may be locked by a single process. Each locked record consumes an entry in the BDOS system Lock List which is shared by locked record and open file entries. Another MP/M-86 system generation parameter sets the size of this table. If there is not sufficient space in the system Lock List to lock all the specified records, or the process record lock limit is exceeded, then the LOCK RECORD function locks no records and returns an Error Code to the calling process.

Upon return, the LOCK RECORD function sets register AL to zero if the lock operation was successful. Otherwise, register AL contains one of the following Error Codes:

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

The function returns Error Code 01 when it accesses a data block that has not been previously written.

The function returns Error Code 03 when it cannot close the current extent prior to moving to a new extent.

The function returns Error Code 04 when it accesses an extent that has not been created.

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

The function returns Error Code 08 if the specified record is locked by another process.

The function returns Error Code 10 if the referenced FCB failed the FCB checksum test.

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

The function returns Error Code 12 when the sum of the number of records currently locked by the calling process and the number of records to be locked by the LOCK RECORD call, exceeds the maximum allowed value. This value is an MP/M-86 GENSYS parameter.

The function returns Error Code 13 when an invalid File ID is placed in the current DMA.

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

01 : Permanent error
04 : Select Error

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

```
*************************************************
*                                               *
*  FUNCTION 43:  UNLOCK RECORD                  *
*                                               *
*************************************************
*                                               *
*        Unlock Records in a Disk File          *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 2BH                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Error Code                 *
*                AH: Physical Error             *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The UNLOCK RECORD function unlocks one or more consecutive records previously locked by the LOCK RECORD function. This function is only supported for files open in Unlocked Mode. If it is called for a file open in Locked or Read/only Mode, no locking action is performed and a successful result is returned.

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

The UNLOCK RECORD function will not unlock a record that is currently locked by another process. However, the function does not return an error if a process attempts to do that. Thus, if the Multi-Sector Count is greater than one, the UNLOCK RECORD function will unlock all records locked by the calling process, while skipping those records locked by other processes.

Upon return, the UNLOCK RECORD function sets register AL to zero if the unlock operation was successful.

Otherwise, register AL contains one of the following Error Codes:

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

The function returns Error Code 01 when it accesses a data block which has not been previously written.

The function returns Error Code 03 when it cannot close the current extent prior to moving to a new extent.

The function returns Error Code 04 when it accesses an extent that has not been created.

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

The function returns Error Code 10 if the referenced FCB failed the FCB checksum test.

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

The functions return Error Code 13 when an invalid File ID is placed in the current DMA.

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

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

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

```
*************************************************
*                                               *
*  FUNCTION 44:  SET MULTI-SECTOR COUNT         *
*                                               *
*************************************************
*                                               *
*     Set Number of Records for Subsequent      *
*             Disk Reads and Writes             *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 2CH                        *
*                DL: Number of Sectors          *
*                                               *
*  Return Values:                               *
*      Register  AL: Return Code                *
*                BL: Same as AL                 *
*                                               *
*************************************************
```

The SET MULTI-SECTOR COUNT function provides logical record blocking under MP/M-86. It enables a process to read and write from 1 to 16 "physical" records of 128 bytes at a time during subsequent BDOS read and write functions. It also specifies the number of 128-byte records to be locked or unlocked by the BDOS LOCK RECORD and UNLOCK RECORD functions.

Function 44 sets the Multi-Sector Count value for the calling process to the value passed in register DL. Once set, the specified Multi-Sector Count remains in effect until the calling process makes another SET MULTI-SECTOR COUNT function call and changes the value. Note that the CLI function sets the Multi-Sector Count to one when it initiates a transient program.

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

Upon return, the function sets register AL to 0 if the specified value is in the range of 1 to 16. Otherwise, it sets register AL to 0FFH.

```
************************************************
*                                              *
*  FUNCTION 45:  SET BDOS ERROR MODE           *
*                                              *
************************************************
*                                              *
* Set BDOS Error Mode for types of Error Returns*
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 2DH                       *
*                DL: BDOS Error Mode           *
*                                              *
************************************************
```

The BDOS Error Mode determines how physical and extended errors (see Section 2.2.13) are handled for a process. The Error Mode can exist in three modes: the default mode, Return Error Mode and Return and Display Error Mode. In the default mode, BDOS displays a system message at the console identifying the error and terminates the calling process. In the return modes, BDOS sets register AL to 0FFH (255 Decimal), places an Error Code identifying the physical or extended error in the four low-order bits of register AH, and returns to the calling process. In Return and Display Mode, the BDOS displays the system message before returning to the calling process. However,when the BDOS is in Return Error Mode, it does not display any system messages.

Function 45 sets the BDOS Error Mode for the calling process to the mode specified in register DL. If register DL is set to 0FFH (255 Decimal), the Error Mode is set to Return Error Mode. If register DL is set to 0FEH (254 Decimal), the Error Mode is set to Return and Display Mode. If register DL is set to any other value, the Error Mode is set to the default mode.

```
************************************************
*                                              *
*  FUNCTION 46:  GET FREE DISK SPACE           *
*                                              *
************************************************
*                                              *
*   Return Free Disk Space on Specified Drive  *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 2EH                       *
*                DL: Drive                     *
*                                              *
*  Return Values:                              *
*      Register  AL: Error Flag                *
*                AH: Physical Error            *
*                BX: Same as AX                *
*                First 3 bytes of DMA buffer   *
*                                              *
************************************************
```

The GET DISK FREE SPACE function determines the number of free sectors (128-byte records) on the specified drive. The calling process passes the drive number in register DL, with 0 for drive A, 1 for B, etc., through 15 for drive P in a full 16-drive system. Function 46 returns a binary number in the first 3 bytes of the current DMA buffer. This number is returned in the format shown in Figure 6-2.

```
+-----+-----+-----+
| fs0 | fs1 | fs2 |
+-----+-----+-----+
```

Figure 6-2. Disk Free Space Field Format

```
fs0 = low    byte
fs1 = middle byte
fs2 = high   byte
```

Upon return, the function sets register AL to 0 if the BDOS Error Mode is the default mode. However, if the BDOS Error Mode is one of the return modes (see Function 45) and a physical error was encountered, it sets register AL to 0FFH (255 Decimal), and register AH to one of the following values:

```
01 - Permanent error
04 - Select error
```

```
*************************************************
*                                               *
*  FUNCTION 47:  CHAIN TO PROGRAM               *
*                                               *
*************************************************
*                                               *
* Load, Initialize and Jump to specified Program*
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 2FH                        *
*                DMA buffer: Command Line       *
*                                               *
*  Return Values:                               *
*      Register  AX: 0FFFFH - Could not find    *
*                             Command           *
*                                               *
*************************************************
```

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

Under MP/M-86, the CHAIN TO PROGRAM function releases the memory of the calling function before executing the command. The command is processed in the same manner as the CLI function (Function 150). If the command warrants the loading of a CMD file and the memory released is large enough for the new program, MP/M-86 loads the new program into the same memory area as the old program.

Except in the case of passing the command to an RSP, the new program is run by the same process that ran the old program. The name of the process is changed to reflect the new program being run. If the command invokes an RSP, the calling process terminates upon successfully writing the command to the RSP queue.

Parameter passing between the old and new programs is accomplished through the use of disk files, queues or the command line. The command line is parsed and placed in the Base Page of the new program in the manner documented in the CLI function (Function 150).

The CHAIN TO PROGRAM function returns an error if there is no RSP with the same name as the command and no CMD file is found. If a CMD file is found and an error occurs after it is successfully opened, the calling process is terminated since its memory has been released.

```
************************************************
*                                              *
*  FUNCTION 48:  FLUSH BUFFERS                 *
*                                              *
************************************************
*                                              *
*        Flush a Write-Deferred Buffers        *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 30H                       *
*                                              *
*  Return Values:                              *
*      Register  AL: Error Flag                *
*                AH: Permanent Error           *
*                BX: Same as AX                *
*                                              *
************************************************
```

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

Upon return, the function sets register AL to 0 if the flush operation was successful. If a physical error was encountered, the FLUSH BUFFERS function performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is in the default mode, the system displays a message at the console identifying the error and terminates the calling process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to the following physical Error Code:

01 : Permanent error

```
*************************************************
*                                               *
*  FUNCTION 50:  DIRECT BIOS CALL               *
*                                               *
*************************************************
*                                               *
*         Call BIOS character routine           *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 32H                        *
*                DX: BIOS Desc. Addr. - Offset  *
*                DS: BIOS Desc. Addr. - Segment *
*                                               *
*  Return Values:                               *
*      Register  AX: BIOS Return                *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

BIOS Descriptor:

```
+-----+-----+-----+-----+-----+
|FUNC |    CX     |    DX     |
+-----+-----+-----+-----+-----+
```

Figure 6-3. BIOS Descriptor Format

The DIRECT BIOS CALL function is provided under MP/M-86 for compatibility with programs generated under CP/M-86 that uses this function. Under MP/M-86, only routines that interface with character devices are supported. The arguments to character routines such as CONIN and LIST are converted to those appropriate for the MP/M-86 XIOS. Where console or list device numbers are needed by the XIOS, default values of the calling process are sent to the XIOS

The FUNC,CX and DX fields of the BIOS Descriptor explained in the Digital Research CP/M-86 System Guide.

```
*************************************************
*                                               *
*  FUNCTION 51:  SET DMA BASE                   *
*                                               *
*************************************************
*                                               *
*    Set Direct Memory Access Segment Address   *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 33H                        *
*                DX: DMA Segment Address        *
*                                               *
*************************************************
```

Function 51 sets the base register for subsequent DMA transfers. The word parameter in DX is a paragraph address and is used with the DMA offset to specify the address of a 128-byte buffer area to be used in the disk read and write functions. Note that upon initial program loading, the default DMA base is set to the address of the user's data segment (the initial value of DS) and the DMA offset is set to 0080H, which provides access to the default buffer in the Base Page.

```
*************************************************
*                                               *
*  FUNCTION 52:  GET DMA ADDRESS                *
*                                               *
*************************************************
*                                               *
* Return Address of Direct Memory Access Buffer *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 34H                        *
*                                               *
*  Return Values:                               *
*      Register  AX: DMA Offset                 *
*                BX: Same as AX                 *
*                ES: DMA Segment                *
*                                               *
*************************************************
```

Function 52 returns the current DMA Base Segment address in ES, with the current DMA Offset in DX.

```
*************************************************
*                                               *
*  FUNCTION 53:  GET MAX MEM                    *
*                                               *
*************************************************
*                                               *
*      Allocate Maximum Memory Available        *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 35H                        *
*                DX: MCB Address - Offset       *
*                DS: MCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: 0 if successful            *
*                    0FFH on failure            *
*                BL: Same as AL                 *
*                CX: Error Code                 *
*                MCB filled in                  *
*                                               *
*************************************************
```

Memory Control Block (MCB):

```
+-----+-----+-----+-----+-----+
|   BASE    |   LENGTH  | EXT |
+-----+-----+-----+-----+-----+
```

Figure 6-4. Memory Control Block Format

BASE — The Segment Address of the beginning of the allocated memory. The function fills in this field on a successful allocation.

LENGTH — Length of the Memory Segment in paragraphs. The LENGTH field is set to the maximum number of paragraphs wanted. The function sets this field to the actual number of paragraphs obtained on a successful allocation.

EXT — The function fills in the EXT byte on a successful allocation and always sets it to one.

Function 53 allocates the largest available memory region which is less than or equal to the LENGTH field of the MCB in paragraphs. If the allocation is successful, the function sets the BASE to the base paragraph address of the available area, and LENGTH to the paragraph length. Upon return, register AL has the value 0FFH if no memory is available, and 00H if the request was successful. The function sets the EXT to 1 if there is additional memory for allocation, and 0 if no additional memory is available.

```
***********************************************
*                                             *
*  FUNCTION 54:  GET ABS MAX                  *
*                                             *
***********************************************
*                                             *
*      Allocate Maximum Memory Available      *
*           at a Specified Address            *
*                                             *
***********************************************
*                                             *
*  Entry Parameters:                          *
*      Register  CL: 36H                      *
*                DX: MCB Address - Offset     *
*                DS: MCB Address - Segment    *
*                                             *
*  Return Values:                             *
*      Register  AL: 0 if successful          *
*                    0FFH on failure          *
*                BL: Same as AL               *
*                CX: Error Code               *
*                MCB filled in                *
*                                             *
***********************************************
```

Memory Control Block (MCB):

```
+-----+-----+-----+-----+-----+
|   BASE    |   LENGTH  | EXT |
+-----+-----+-----+-----+-----+
```

Figure 6-4. Memory Control Block Format

BASE The Segment Address of the beginning of the memory segment wanted. This field is maintained on a successful allocation.

LENGTH Length of the Memory Segment in paragraphs. The LENGTH field is set to the maximum number of paragraphs wanted. On a successful allocation, the function sets this field to the actual number of paragraphs obtained.

EXT The EXT field is unused but must be available.

Function 54 is used to allocate the largest possible region at the absolute paragraph boundary given by the BASE field of the MCB, for a maximum of LENGTH paragraphs. If the allocation is successful, the function sets the LENGTH to the actual length. Upon return, register AL has the value 0FFH if no memory is available at the absolute address, and 00H if the request was successful.

```
***********************************************
*                                             *
*  FUNCTION 55:  ALLOC MEM                    *
*                                             *
***********************************************
*                                             *
*          Allocate a Memory Segment          *
*                                             *
***********************************************
*                                             *
*  Entry Parameters:                          *
*      Register  CL: 37H                      *
*                DX: MCB Address - Offset     *
*                DS: MCB Address - Segment    *
*                                             *
*  Return Values:                             *
*      Register  AL: 0 if successful          *
*                    OFFH on failure          *
*                BL: Same as AL               *
*                CX: Error Code               *
*                MCB filled in                *
*                                             *
***********************************************
```

Memory Control Block (MCB):

```
+-----+-----+-----+-----+-----+
|   BASE    |   LENGTH  | EXT |
+-----+-----+-----+-----+-----+
```

Figure 6-5. Memory Control Block Format

BASE — The Segment Address of the beginning of the memory segment allocated. The function fills in this field on a successful allocation.

LENGTH — Length of the Memory Segment in paragraphs. The LENGTH field is set to the number of paragraphs wanted. On a successful allocation, this field is maintained.

EXT — The EXT field is unused but must be available.

The ALLOCATE MEMORY function allocates a memory area whose size is the LENGTH field of the MCB. Function 55 returns the base paragraph address of the allocated region in the user's MCB. Upon return, register AL contains a 00H if the request was successful and a 0FFH if the memory could not be allocated.

```
*************************************************
*                                               *
*  FUNCTION 56:  ALLOC ABS MAX                  *
*                                               *
*************************************************
*                                               *
*          Allocate a Memory Segment            *
*            at a Specified Address             *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 38H                        *
*                DX: MCB Address - Offset       *
*                DS: MCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: 0 if successful            *
*                    0FFH on failure            *
*                BL: Same as AL                 *
*                CX: Error Code                 *
*                MCB filled in                  *
*                                               *
*************************************************
```

Memory Control Block (MCB):

```
+-----+-----+-----+-----+-----+
|   BASE    |   LENGTH  | EXT |
+-----+-----+-----+-----+-----+
```

Figure 6-6. Memory Control Block Format

BASE — The Segment Address of the beginning of the memory segment wanted. This field is maintained on a successful allocation.

LENGTH — Length of the Memory Segment in paragraphs. The LENGTH field is set to the number of paragraphs wanted. This field is maintained on a successful allocation.

EXT — The EXT field is unused but must be available.

The ALLOCATE ABSOLUTE MEMORY function allocates a memory area which starts at the address specified by the BASE field and whose length is specified by the LENGTH field of the MCB. Upon return, register AL contains a 00H if the request was successful and a 0FFH if the memory could not be allocated.

```
*************************************************
*                                               *
*  FUNCTION 57:  FREE MEM                       *
*                                               *
*************************************************
*                                               *
*        Free a specified Memory Segment        *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 39H                        *
*                DX: MCB Address - Offset       *
*                DS: MCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: 0 if successful            *
*                    0FFH on failure            *
*                BL: Same as AL                 *
*                CX: Error Code                 *
*                MCB filled in                  *
*                                               *
*************************************************
```

```
           Memory Control Block (MCB):

        +-----+-----+-----+-----+-----+
        |   BASE    |   LENGTH  | EXT |
        +-----+-----+-----+-----+-----+
```

Figure 6-7. Memory Control Block Format

BASE — A Segment Address in the memory segment which begins the area to be freed.

LENGTH — Length of the Memory Segment in paragraphs. This field is not used. The memory area freed always goes to the end of the previously allocated memory segment.

EXT — If the EXT field is 00H, the memory segment specified by the BASE field is freed. If the value is 0FFH, all memory except memory allocated at load time is freed.

The FREE MEMORY function is used to release memory areas allocated to the program. The value of the EXT field of the MCB controls the operation of this function. If EXT = 0FFH then the function releases all memory areas allocated by the calling program. If the EXT field is zero, the function releases the memory area beginning at the specified BASE and ending at the end of the previously allocated memory segment.

```
*************************************************
*                                               *
*  FUNCTION 58:  FREE ALL MEM                   *
*                                               *
*************************************************
*                                               *
*          Terminate Calling Process            *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 3AH                        *
*                                               *
*************************************************
```

In the CP/M-86 environment, the FREE ALL MEMORY function releases all memory in order to release memory allocated by interface type programs before returning to the CCP. Under MP/M-86, the equivalent action is to terminate the calling process.

```
*************************************************
*                                               *
*  FUNCTION 59:  PROGRAM LOAD                   *
*                                               *
*************************************************
*                                               *
*          Load a Progam into Memory            *
*             From a CMD type file              *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 3BH                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AX: Base Page Segment          *
*                    OFFFFH on error            *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The PROGRAM LOAD function loads a CMD type disk file into memory. Upon entry, register DX contains the DS-relative offset of a successfully opened FCB that names the input CMD file. Upon return, register AX has the value OFFFFH if the program load was unsuccessful. Otherwise, AX contains the paragraph address of the Base Page belonging to the loaded program. The base address and segment length of each segment is stored in the Base Page. Upon program load, the CLI function initializes the DMA base address to the Base Page of the loaded rogram, and the DMA offset address to 0080H. Note: the CLI function performs this initialization. The PROGRAM LOAD function does not establish a default DMA address. A program must execute Function 51 (SET DMA BASE) and Function 26 (SET DMA OFFSET) before executing the PROGRAM LOAD function. If a new process is to run the loaded program, it must initialize a User Data Area (UDA) and a Process Descriptor (PD) and then call the CREATE PROCESS function. It is recommended that the SEND CLI COMMAND function be used in the case of creating a new process.

```
**************************************************
*                                                *
*  FUNCTION 100:  SET DIRECTORY LABEL            *
*                                                *
**************************************************
*                                                *
*      Create or Update a Directory Label        *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 64H                         *
*                DX: FCB Address - Offset        *
*                DS: FCB Address - Segment       *
*                                                *
*  Return Values:                                *
*      Register  AL: Directory Code              *
*                AH: Physical or Extended Error  *
*                BX: Same as AX                  *
*                                                *
**************************************************
```

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

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

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

Upon return, Function 100 returns a Directory Code in register AL with the value 0 to 3 if the Directory Label create or update was successful, or 0FFH (255 Decimal) if no space existed in the referenced directory to create a Directory Label. Register AH is set to zero in both of these cases. If a physical or extended error was encountered, Function 100 performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is the default mode, the system displays a message at the console identifying the error and terminates the calling process. Otherwise, Function 100 returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended Error Codes:

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

```
*************************************************
*                                               *
*  FUNCTION 101:  RETURN DIRECTORY LABEL        *
*                                               *
*************************************************
*                                               *
*      Return Data Byte of Directory Label      *
*            for the specified Drive            *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 65H                        *
*                DL: Drive                      *
*                                               *
*  Return Values:                               *
*      Register  AL: Directory Label Data Byte  *
*                AH: Physical Error             *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

The RETURN DIRECTORY LABEL function returns the data byte of the Directory Label for the specified drive. The calling process passes the drive number in register E with 0 for drive A, 1 for drive B, and so-forth through 15 for drive P in a full 16-drive system. The format of the Directory Label data byte is shown below:

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

Function 101 returns the Directory Label data byte to the calling process in register AL. Register AL equal to 0 indicates that no Directory Label exists on the specified drive. If the function encounters a physical error when the BDOS Error mode is in one of the return modes (see Function 45), it returns with register AL set to 0FFH (255 Decimal) and register AH set to one of the following:

01 : Permanent error
04 : Select error

```
*************************************************
*                                               *
*  FUNCTION 102:  READ FILE XFCB                *
*                                               *
*************************************************
*                                               *
*      Return Extended File Control Block       *
*               of a Disk File                  *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 66H                        *
*                DX: FCB Address - Offset       *
*                DS: FCB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AL: Directory Code             *
*                AH: Physical Error             *
*                BX: Same as AX                 *
*                                               *
*************************************************
```

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

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

```
byte 12      : XFCB password mode field

                  bit 7 - Read mode
                  bit 6 - Write mode
                  bit 5 - Delete mode

     Byte 12 equal to 0 indicates the file has not
               been assigned a password.

byte 13 - 23 : XFCB password field (encrypted)
byte 24 - 27 : XFCB Create or Access time stamp field
byte 28 - 31 : XFCB Update time stamp field
```

Upon return, Function 102 returns a Directory Code in register AL with the value 0 to 3 if the XFCB read operation was successful, or 0FFH (255 Decimal) if the XFCB was not found. Register AH is set to zero in both of these cases. If a physical or extended error was encountered, Function 102 performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is in the default mode, the system displays a message at the console identifying the error and terminates the calling process.

Otherwise, Function 102 returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical Error Codes:

01 : Permanent error
04 : Select Error

```
************************************************
*                                              *
*  FUNCTION 103:  WRITE FILE XFCB              *
*                                              *
************************************************
*                                              *
*       Write Extended File Control Block      *
*                of a Disk File                *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 67H                       *
*                DX: FCB Address - Offset      *
*                DS: FCB Address - Segment     *
*                                              *
*  Return Values:                              *
*      Register  AL: Directory Code            *
*                AH: Physical or Extended Error *
*                BX: Same as AX                *
*                                              *
************************************************
```

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

```
 FCB byte 12 (ex) : XFCB password mode

bit 7 - Read mode
bit 6 - Write mode
bit 5 - Delete mode
bit 0 - assign new password to the file
```

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

Upon return, Function 100 returns a Directory Code in register AL with the value 0 to 3 if the XFCB create or update was successful, or 0FFH (255 Decimal) if no Directory Label existed on the specified drive, or the file named in the FCB was not found, or no space existed in the directory to create an XFCB. Register AH is set to zero in all of these cases. If a physical or extended error was encountered, Function 103 performs different actions depending on the BDOS Error Mode (see Function 45). If the BDOS Error Mode is the default mode, the system displays a message at the console identifying the error and terminates the calling process. Otherwise, Function 103 returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended Error Codes:

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

```
*************************************************
*                                               *
*  FUNCTION 104:  SET DATE AND TIME             *
*                                               *
*************************************************
*                                               *
*            Set System Date and Time           *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 68H                        *
*                DX: TOD Address - Offset       *
*                DS: TOD Address - Segment      *
*                                               *
*************************************************
```

The SET DATE AND TIME function sets the system internal date and time. The calling process passes the address of a 4-byte structure containing the date and time specification. The format of the date and time data structure is:

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

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

Under MP/M-86, this function also sets the Second field of the system date and time to zero.

```
**************************************************
*                                                *
*  FUNCTION 105:  GET DATE AND TIME              *
*                                                *
**************************************************
*                                                *
*            Set System Date and Time            *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 69H                         *
*                DX: TOD Address - Offset        *
*                DS: TOD Address - Segment       *
*                                                *
*  Return Values:                                *
*                TOD filled in                   *
*                                                *
**************************************************
```

The GET DATE AND TIME function obtains the system internal date and time. The calling process passes the address of a four-byte data structure which receives the date and time values. The format of the data structure is the same as the format described in Function 104. This function is equivalent to MP/M-86 Function 155 except that it does not return the seconds field of the internal time.

```
**************************************************
*                                                *
*  FUNCTION 106:  SET DEFAULT PASSWORD           *
*                                                *
**************************************************
*                                                *
*  Establish a Default Password for file access *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 6AH                         *
*                DX: Password Address - Offset   *
*                DS: Password Address - Segment *
*                                                *
**************************************************
```

The SET DEFAULT PASSWORD function allows a process to specify a password value before a process accesses a file protected by the password. When the file system accesses a password protected file, it checks the current DMA and the default password for the correct value. The function does not return a password error if either password is correct. MP/M-86 maintains a default password for each process currently running on the system. When a process (parent) creates a subprocess (child), the child process inherits its default console from its parent. Note: changing the default password does not affect other processes currently running on the system.

To make a Function 106 call, the calling process passes the address of an eight-byte field containing the password.

```
*************************************************
*                                               *
*  FUNCTION 107:  RETURN SERIAL NUMBER          *
*                                               *
*************************************************
*                                               *
*   Return the Current System's Serial Number   *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 6BH                        *
*                DX: SERIAL Address - Offset    *
*                DS: SERIAL Address - Segment   *
*                                               *
*  Return Values:                               *
*                SERIAL filled in               *
*                                               *
*************************************************
```

Function 107 returns the MP/M-86 serial number to the addressed, six-byte SERIAL field.

```
*************************************************
*                                               *
*  FUNCTION 128:  MEMORY ALLOCATION             *
*  FUNCTION 129:                                *
*                                               *
*************************************************
*                                               *
*           Allocate a Memory Segment           *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 080H or 081H               *
*                DX: MPB Address-Offset         *
*                DS: MPB Address-Segment        *
*                                               *
*  Return Values:                               *
*      Register  AX: 0 (success)                *
*                    0ffffH (fail)              *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|  START    |    MIN    |    MAX    |   0000    |   0000    |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-7. Memory Parameter Block (MPB)

START	if non-zero, an absolute request at this paragraph
MIN	minimum memory needed (paragraphs)
MAX	maximum memory wanted (paragraphs)
0000	these fields must be zero (0). They are used internally and for future use.

The MEMORY ALLOCATION function allows a program to allocate extra memory. A successful allocation will allocate a contiguous memory segment whose length is at least the MIN and no more than the MAX number of paragraphs specified in the MPB. The START field of the MPB is modified to be the starting paragraph of the memory segment. The MIN and MAX fields are modified to be the length of the memory segment in paragraphs. Memory Segments can be explicitly released through the MEMORY FREE function. MP/M-86 will also release all memory owned by a process at termination.

```
**************************************************
*                                                *
*  FUNCTION 130:  MEMORY FREE                    *
*                                                *
**************************************************
*                                                *
*              Free a Memory Segment             *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 082H                        *
*                DX: MFPB Address - Offset       *
*                DS: MFPB Address - Segment      *
*  Return Values:                                *
*      Register  AX: 0 on success                *
*                    0ffffH on failure           *
*                BX: Same as AX                  *
*                CX: Error Code                  *
*                                                *
**************************************************
```

```
+-----+-----+-----+-----+
|   START   |   0000    |
+-----+-----+-----+-----+
```

Figure 6-8. Memory Free Parameter Block (MFPB)

The MEMORY FREE function releases memory starting at the START paragraph to the end of a single previously allocated segment that contains the START paragraph. If the START paragraph is the same as that returned in the MPB of a memory allocation call, then Function 130 releases the whole memory segment.

Under certain circumstances, MP/M-86 allows memory segments to be shared among different processes. In this case, the system recovers a released memory segment only when no other processes own the memory segment.

```
*************************************************
*                                               *
*  FUNCTION 131:  POLL DEVICE                   *
*                                               *
*************************************************
*                                               *
*                   Poll a Device               *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 083H                       *
*                DL: Device Number              *
*                                               *
*  Return Values:                               *
*      Register  AX: 0 on success               *
*                    0ffffH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The POLL DEVICE function is used by the XIOS to poll non-interrupt driven devices. It is should be used whenever the XIOS is waiting for a non-interrupt event. The calling process relinquishes the CPU and allows MP/M-86 to poll the device at every dispatch. The XIOS contains routines for each device number. These routines are called through the XIOS POLL DEVICE function (see the description in the MP/M-86 System Guide), and they return whether the device is ready or not. When the device is ready, MP/M-86 will restore the calling process to the 'RUN' state and return. Upon return, the calling process knowns that the device is ready.

```
*************************************************
*                                               *
*  FUNCTION 132:  FLAG WAIT                     *
*                                               *
*************************************************
*                                               *
*              Wait for a System Flag           *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 084H                       *
*                DL: Flag Number                *
*                                               *
*  Return Values:                               *
*      Register  AX: 0 on success               *
*                    0ffffH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The FLAG WAIT function is used by a process to wait for an interrupt. The process relinquishes the CPU until an interrupt routine calls the FLAG SET function which places the waiting process in the 'RUN' state. When Function 132 returns to the calling process, the interrupt has either occured, or an error occured. An error occurs when a process is already waiting for the flag. If the Flag was set before Function 132 was called, the routine returns successfully without relinquishing the CPU. This routine is meant to be used by the XIOS. The mapping between types of interrupts and flag numbers is maintained in the XIOS, although MP/M-86 reserves flags 0,1,2 and 3 for system use.

```
*************************************************
*                                               *
*  FUNCTION 133:  FLAG SET                      *
*                                               *
*************************************************
*                                               *
*                 Set a System Flag             *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 085H                       *
*                DL: Flag Number                *
*                                               *
*  Return Values:                               *
*      Register  AX: 0 on success               *
*                    0ffffH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The FLAG SET function is used by interrupt routines to notify the system that a logical interrupt has occured. A process waiting for this flag will be placed back into the 'RUN' state. If there are no processes waiting, then the next process to wait for this flag will return successfully without relinquishing the CPU. The function detects an error if the flag has already been set and no process has done a FLAG WAIT call to reset it.

```
***********************************************
*                                             *
*  FUNCTION 134:  MAKE QUEUE                  *
*                                             *
***********************************************
*                                             *
*               Make a System Queue           *
*                                             *
***********************************************
*                                             *
*  Entry Parameters:                          *
*      Register  CL: 086H                     *
*                DX: QD Address - Offset      *
*                DS: QD Address - Segment     *
*                                             *
*  Return Values:                             *
*      Register  AX: 0 on success             *
*                    0ffffH on failure        *
*                BX: Same as AX               *
*                CX: Error Code               *
*                                             *
***********************************************
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|  * 0000h  |  * 0000h  |   FLAGS   | NAME ...  
+-----+-----+-----+-----+-----+-----+-----+-----+
|         ... NAME                  |  MSGLEN   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  NMSGS    |  * 0000h  |  * 0000h  |  * 0000h  |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  * 0000h  |  BUFFER   |
+-----+-----+-----+-----+
```

Figure 6-8. Queue Descriptor (QD) Format

FLAGS Queue Flags. The bits are defined as follows:

0001H - Mutual Exclusion Queue
0002H - CANNOT be deleted
0004H - restricted to SYSTEM processes
0008H - RSP message queue
0010H - Used Internally
0020H - RPL address queue
0040H - Used Internally
0080H - Used Internally

Remaining Flags reserved for future use

NAME 8-byte Queue Name. All 8 bits of each character are matched on an OPEN QUEUE call.

MSGLEN Number of bytes in each logical message

NMSGS Maximum number of logical messages to be supported.

	If the number of messages written to the queue equals this maximum, no more messages are allowed until a message is read.
BUFFER	address of the queue buffer. This buffer must be (NMSGS * MSGLEN) bytes long. The address is an offset relative to the DS register. This field is unused if the QD resides outside of the System Data Area. Typically this field is 0 if the queue is being created by a transient program. RSPs that create queues _must_ initialize this field to point to a buffer. The Data Segment of an RSP's queue is considered part of the System Data Area unless it is beyond 64k of the beginning of the System Data Area.
0000	for internal use. Must be initialized to zero.

Every System Queue under MP/M-86 is associated with a Queue Descriptor that resides within the MP/M-86 System Data Area. In the MAKE QUEUE function, the calling process passes the address of a Queue Descriptor. If this Queue Descriptor is within the MP/M-86 System Data Area, the system uses it directly for the System Queue. If the Queue Descriptor is outside of the System Data Area, the system obtains a Queue Descriptor from an internal Queue Descriptor table. If there are no unused Queue Descriptors in the internal table, the function returns an Error Code. The size of this table is determined by GENSYS at system generation time.

The buffer for a System Queue must also reside within the System Data area. For non-zero length buffers, resident buffers are used directly. The system obtains a buffer from the Queue Buffer Area if the buffer does not reside within the System Data Area. The size of the buffer is calculated from the NMSGS and MSGLEN fields. The function returns an Error Code if there is not enough unused buffer area left to accommodate this new buffer. The size of the Queue Buffer Area is determined by GENSYS at system generation time.

All System Queues must have unique names. The function returns an Error Code if a System Queue already exists by the given name.

Under MP/M-86, all System Queues must be explicitly opened (see Function 135) before being used to read or write messages or to delete the queue.

```
*************************************************
*                                               *
*  FUNCTION 135:  OPEN QUEUE                    *
*                                               *
*************************************************
*                                               *
*              Open a System Queue              *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 087H                       *
*                DX: QPB Address - Offset       *
*                DS: QPB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AX: 0 on success               *
*                    0ffffH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|  RESERVED |  QUEUEID  |   NMSGS   |  BUFFER   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                     NAME                      |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-9. Queue Parameter Block (QPB)

RESERVED	must be zero, modified by OPEN QUEUE
QUEUEID	modified by OPEN QUEUE
NMSGS	not used for OPEN QUEUE
BUFFER	not used for OPEN QUEUE
NAME	8-byte System Queue name.

All System Queues under MP/M-86 must be explicitily opened before a read, write or delete operation can be done. The OPEN QUEUE function examines each existing System Queue and attempts to match the name in the QPB with the name of a System Queue. All eight bytes of the name must match for a successful open. All bits of each byte are examined. If the open operation is successful, the OPEN QUEUE function modifies the QUEUE ID Field of the QPB. Once the the Queue is opened, subsequent reads, writes or a delete are allowed.

The function returns an Error Code if the System Queue does not exist, or if it is restricted to SYSTEM processes and the calling process is a USER process.

```
*************************************************
*                                               *
*  FUNCTION 136:  DELETE QUEUE                  *
*                                               *
*************************************************
*                                               *
*              Delete a System Queue            *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 088H                       *
*                DX: QPB Address - Offset       *
*                DS: QPB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AX: 0 on success               *
*                    0ffffH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|  RESERVED |  QUEUEID  |   NMSGS   |  BUFFER   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                     NAME                      |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-10. Queue Parameter Block (QPB)

RESERVED	filled in by previous OPEN QUEUE
QUEUEID	filled in by a previous OPEN QUEUE
NMSGS	not used for DELETE QUEUE
BUFFER	not used for DELETE QUEUE
NAME	not used for DELETE QUEUE

The DELETE QUEUE function removes a System Queue from the system. The system returns Error Codes if the Queue cannot be deleted or if the Queue hasn't been previously opened.

```
************************************************
*                                              *
*  FUNCTION 137:  READ QUEUE                   *
*                                              *
************************************************
*                                              *
*       Read a Message from a System Queue     *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 089H                      *
*                DX: QPB Address - Offset      *
*                DS: QPB Address - Segment     *
*                                              *
*  Return Values:                              *
*      Register  AX: 0 on success              *
*                    0ffffH on failure         *
*                BX: Same as AX                *
*                CX: Error Code                *
*                                              *
************************************************
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| RESERVED  | QUEUEID   |   NMSGS   |  BUFFER   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                    NAME                       |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-11. Queue Parameter Block (QPB)

RESERVED filled in by previous OPEN QUEUE

QUEUEID filled in by previous OPEN QUEUE

NMSGS number of messages to read

BUFFER offset of buffer relative to the current Data Segment. Message is placed in buffer indicated.

NAME not used by READ QUEUE

The READ QUEUE function reads a message from a System Queue that was previously opened by the calling process. The function returns an Error Code if the Queue was not previously opened or if the System Queue has been deleted since the OPEN QUEUE call. If the NMSGS field is zero (0) or one (1), then the function reads one message and places it into the buffer indicated by the BUFFER field of the QPB. If there are not enough messages to read from the Queue, the calling process waits until another process writes into the queue before returning.

```
*************************************************
*                                               *
*  FUNCTION 138:  CONDITIONAL READ QUEUE        *
*                                               *
*************************************************
*                                               *
*         Conditionally Read a Message          *
*             from a System Queue               *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 08aH                       *
*                DX: QPB Address - Offset       *
*                DS: QPB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AX: 0 on success               *
*                    0ffffH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| RESERVED  |  QUEUEID  |  NMSGS    |  BUFFER   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                     NAME                      |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-12. Queue Parameter Block (QPB)

RESERVED	filled in by previous OPEN QUEUE
QUEUEID	filled in by previous OPEN QUEUE
NMSGS	number of messages to read
BUFFER	offset of buffer relative to the current Data Segment. Message is placed in buffer indicated.
NAME	not used by READ QUEUE

The CONDITIONAL READ QUEUE function is analagous to the READ QUEUE function except that it returns an Error Code if there are not enough messages to read instead of waiting for another process to write to the queue.

```
************************************************
*                                              *
*  FUNCTION 139:  WRITE QUEUE                  *
*                                              *
************************************************
*                                              *
*        Write a Message to a System Queue     *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 08bH                      *
*                DX: QPB Address - Offset      *
*                DS: QPB Address - Segment     *
*                                              *
*  Return Values:                              *
*      Register  AX: 0 on success              *
*                    0ffffH on failure         *
*                BX: Same as AX                *
*                CX: Error Code                *
*                                              *
************************************************
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| RESERVED  | QUEUEID   |   NMSGS   |  BUFFER   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                    NAME                       |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-13. Queue Parameter Block (QPB)

RESERVED filled in by previous OPEN QUEUE

QUEUEID filled in by previous OPEN QUEUE

NMSGS number of messages to write

BUFFER offset of buffer relative to the current Data Segment. Message is read from buffer indicated.

NAME not used by WRITE QUEUE

The WRITE QUEUE function writes a message to a System Queue that was previously opened by the calling process. The function returns an Error Code if the Queue was not previously opened or if the System Queue has been deleted since the OPEN QUEUE call. If the NMSGS field is zero (0) or one (1), then the function reads one message from the buffer indicated by the BUFFER field of the QPB and writes it into the System Queue Buffer. If there is not enough buffer space in the Queue, the calling process waits until another process reads from the queue before writing to the Queue and returning.

```
*************************************************
*                                               *
*  FUNCTION 140:  CONDITIONAL WRITE QUEUE       *
*                                               *
*************************************************
*                                               *
*          Conditionally Write a Message        *
*                to a System Queue              *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 08cH                       *
*                DX: QPB Address - Offset       *
*                DS: QPB Address - Segment      *
*                                               *
*  Return Values:                               *
*      Register  AX: 0 on success               *
*                    0ffffH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| RESERVED  | QUEUEID   |   NMSGS   |  BUFFER   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                     NAME                      |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-14. Queue Parameter Block (QPB)

RESERVED filled in by previous OPEN QUEUE

QUEUEID filled in by previous OPEN QUEUE

NMSGS number of messages to write

BUFFER offset of buffer relative to the current Data Segment. Message is read from buffer indicated.

NAME not used by WRITE QUEUE

The CONDITIONAL WRITE QUEUE function performs ia analagous to the WRITE QUEUE function except that it returns an Error Code if there is not enough System Queue Buffer to for the message to be written instead of waiting for another process to read from the queue.

```
**************************************************
*                                                *
*  FUNCTION 141:  DELAY                          *
*                                                *
**************************************************
*                                                *
*   Delay for specified number of System Ticks   *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 08dH                        *
*                DX: Number of System Ticks      *
*                                                *
**************************************************
```

The DELAY function causes the calling process to wait a until the specified number of System Ticks has occured. The DELAY function avoids the necessity of programmed delay loops. It allows other processes to use the CPU resource while the calling process waits.

The length of the System Tick varies among installations. A typical System Tick is 60Hz (16.67 milliseconds). In Europe, it is likely to be 50Hz (20 milliseconds). The exact length of the System Tick can be obtained by reading the 'TICKSPERSEC' value from the System Data Area. (see the MP/M-86 System Guide).

There is up to one Tick of uncertainty in the exact amount of time delayed. This is due to the DELAY function being called asynchronously from the actual time base. The DELAY function is guaranteed to delay the calling process at least the number of ticks specified. However, when the calling process is rescheduled to run, it may wait quite a bit longer if there are higher priority processes waiting to run. The DELAY function is used primarily by programs that need to wait specific amounts of time for I/O events to occur. Under these conditions, the calling process usually has a very high priority level. If a process with a high priority calls the DELAY function, the actual delay is typically within a System Tick of the amount of time wanted.

```
*************************************************
*                                               *
*  FUNCTION 142:  DISPATCH                      *
*                                               *
*************************************************
*                                               *
*             Call System Dispatcher            *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 08eH                       *
*                                               *
*************************************************
```

The DISPATCH function forces a reschedule of processes that are waiting to run. Normally, dispatches occur at every interrupt, and whenever a process releases a system resource. Dispatching also occurs whenever a process needs a system resource that is not currently available. For a CPU-bound process, dispatch occurs at the next System Tick.

The MP/M-86 Dispatcher is priority driven, with round-robin scheduling of equivalent-priority processes. When a process calls the DISPATCH function, it is rescheduled process such that processes with higher or equivalent priorities are given the CPU before the calling process obtains it again.

```
************************************************
*                                              *
*  FUNCTION 143:  TERMINATE                    *
*                                              *
************************************************
*                                              *
*           Terminate Calling Process          *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 08fH                      *
*                DL: Terminate Code            *
*                                              *
************************************************
```

The TERMINATE function terminates the calling process. If the Terminate Code is not 0ffH, the function can only terminate a USER process. If the Terminate Code is 0ffH, the function can terminate the calling process even though the process's SYSTEM flag is on. Function 143 can not terminate a process with the KEEP flag on. If the termination is successful, the function releases the Mutual Exclusion Queues owned by the process. It also releases all memory segments owned by the process, and returns the Process Descriptor to the PD table. Since memory can be owned by more than one process, the system does not recover memory segments system until every process owning the memory segment has either terminated or explicitly releases the memory segment with the MEMORY FREE call.

Function 143 does not return any results to the calling process. If the function returns to the calling process then the TERMINATE call failed for one of two reasons. Either the process has the KEEP flag on, or it has the SYSTEM flag on, and the Terminate Code is not 0ffH.

```
*************************************************
*                                               *
*  FUNCTION 144:  CREATE PROCESS                *
*                                               *
*************************************************
*                                               *
*                  Create a Process             *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 090H                       *
*                DX: PD Address - Offset        *
*                DS: PD Address - Segment       *
*                                               *
*  Return Values:                               *
*      Register  AX: 0 on success               *
*                    0ffffH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The CREATE PROCESS function allows a process to create a subprocess within its own memory area. The child process shares all memory owned by the calling process at the time of the CREATE PROCESS call. If the Process Descriptor (PD) is outside of the Operating System Area, the system copies it into a PD from the internal PD Table. The function returns an Error Code if there are no more unused PDs in the Table. The number of PDs in the PD Table is specified by GENSYS at system generation time. The User Data Area (UDA) can be anywhere in memory but is required to be on a paragraph boundary. A Resident System Process (RSP) is considered within the Operating System Area. PDs that reside within an RSP are usually not copied into the PD Table. The only time the system copies the PD is if it is not within 64k of the System Data Area. Process Descriptors as well as Queue Descriptors and Queue Buffers are required to be within the System Data Area because they are linked together on various System Lists or are used by more than one process. Because of this, they cannot be in the Transient Process Area (TPA) where they cannot be protected.

More than one process can be created by a single CREATE PROCESS call if the LINK field of the PD is non-zero. In this case, it is assumed to point to another PD within the same Data Segment. After it creates the first process, the function checks the Process Descriptors LINK field. Using this linked list of PDs, a single CREATE PROCESS call can create multiple processes.

NOTE!! The function does not check the validity of the PD addresses passed by the calling process. An invalid PD address can cause MP/M-86 to crash if no hardware memory protection is available on the system.

```
   +-----+-----+-----+-----+-----+-----+-----+-----+
00 |   LINK    |   THREAD  |STAT |PRIOR|   FLAG    |
   +-----+-----+-----+-----+-----+-----+-----+-----+
08 |                     NAME                      |
   +-----+-----+-----+-----+-----+-----+-----+-----+
10 |    UDA    |DISK | USER|        RESERVED       |
   +-----+-----+-----+-----+-----+-----+-----+-----+
18 |              RESERVED             |  PARENT   |
   +-----+-----+-----+-----+-----+-----+-----+-----+
20 | CNS |     RESERVED    |LIST |    RESERVED     |
   +-----+-----+-----+-----+-----+-----+-----+-----+
28 |                   RESERVED                    |
   +-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-14. Process Descriptor (PD) Format

LINK — link field for insertion on current System List. If this fields initial value is non-zero, it is assumed to point to another PD. This field is used to create more than one process with a single CREATE PROCESS call.

THREAD — link field for insertion on Thread List. Initialized to be zero (0).

STAT — Current Process activity. Initialized to be zero (0).

00 RUN	The process is ready to run. The STAT field will always be in this state when a process is examining its own Process Descriptor. The PD is on the Ready List. The Currently running process is always at the head of Ready List.
01 POLL	The process is polling a device. The PD is on the Poll List.
02 DELAY	The process is delaying for a specified number of System Ticks. The PD is on the Delay List
06 DQ	The process is waiting to read a message from a System Queue that is empty. The PD is on the DQ List whose root is in the Queue Descriptor of the System Queue involved.
07 NQ	The process is waiting to write a message to a System Queue whose buffer is full. The PD is on the NQ List whose root is in the Queue Descriptor of the System Queue involved.
08 FLAGWAIT	The process is waiting for a System Flag to be set. The PD is in the Flag Table entry of the flag it is waiting for.
09 CIOWAIT	The process is waiting to attach to a Character I/O device (console or list) while another process owns it. The PD is on CQUEUE List whose root is in the Character Control Block of the Device in question.

PRIOR — current priority. Process scheduling is done based on this field. Typical user programs run at a priority of

200. 0 is the 'best' priority and 255 is the 'worst' priority. The following is a list of priority assignments used by most MP/M-86 systems.

0 - 31	Interrupt handlers
32 - 63	System processes
64 - 149	Undefined
150	Initialization Process
151 - 197	Undefined
198	Terminal Message Process
199	Undefined
200	Default User Priority
201 - 254	User Processes
255	Idle Process

FLAG Bit field of flags determining run time characteristics of a process. Initialize as needed. All undocumented flags are used internally or are reserved for future use.

001H
SYS System Process. Has priviledged access to various features of MP/M-86. This process can only be terminated if the Termination Code is 0ffh. This process can access restricted System Queues. This flag is turned off if the calling process is not a System Process.

002H
KEEP This process cannot be terminated. This flag is turned off if the calling process is not a System Process.

004H
KERNEL This process resides within the Operating System. This flag is turned off if the PD is not within the Operating System.

010H
TABLE This PD originated from the PD Table. When this process terminates, the PD will be recycled into the PD Table.

020H
RESOURCE This process is currently waiting for a resource. Set to zero at initialization.

040H
RAW This process is doing RAW Character I/O through its default console. Reset at each Console Call.

080H
↑C An attempt was made to terminate this process through some external event but could not be terminated because of the TEMPKEEP flag. Initialized to zero.

NAME Process Name. Eight bytes, all eight bits of each byte is used for matching process names.

UDA Segment Address of this processes User Data Area. Initialized to be the number of paragraphs from the beginning of the calling processes Data Segment. The User Data Area contains process information that is not needed between processes. It also contains the System Stack of each process. See UDA description Below.

DISK	Current default disk
USER	Current default user number
PARENT	process that created this process.
CNS	Current default console's Character Control Block. Initialized to be the default console number.
LIST	Current default List device's Character Control Block. Initialized to be the default List device number.
RESERVED	Reserved for internal Use. These fields must be initialized to zero (0).

```
     +-----+-----+-----+-----+-----+-----+-----+-----+
 00h |  RESERVED | DMA OFFSET|        RESERVED         |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 08h |                   RESERVED                    |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 10h |                   RESERVED                    |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 18h |                   RESERVED                    |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 20h |    AX     |    BX     |    CX     |    DX     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 28h |    DI     |    SI     |    BP     | RESERVED  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 30h |       RESERVED        |    SP     | RESERVED  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 38h |         INT 0         |         INT 1         |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 40h |         INT 2         |         INT 3         |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 48h |         INT 4         |       RESERVED        |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 50h |    CS     |    DS     |    ES     |    SS     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 58h |        INT 224        |        INT 225        |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 60h |                   RESERVED                    |
     +-----+-----+-----+-----+-----+-----+-----+-----+
 68h |                                               | 6Fh
  .  |                                               |
  .  |      U S E R   S Y S T E M   S T A C K        |
  .  |                                               |
 F8h |                                               | FFh
     +-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-15. User Data Area (UDA)

The length of the UDA is 256 bytes,and it must begin on a paragraph boundary.

DMA OFFS — The initial DMA offset for the new process. The segment address of the DMA is assumed to be the same as the initial Data Segment (see DS below).

AX,BX,CX,DX, DI,SI,BP — The Initial register values for the new process. These are typically set to zero.

SP — The initial Stack Pointer for the new process. The Stack Pointer is relative to the initial Stack Segment (see SS Below). The initial stack of the new process must be initialized with the offset of the first instruction to be executed by the new process. The word that the Stack Pointer points to is the initial Instruction Pointer. Two words must follow the initial IP which will be filled in with the initial Code Segment (see CS Below) and the initial flags. The Initial flags will be set to 0200H which means that interrupts are on and all other flags are off. MP/M-86 starts a new process by executing an Interrupt Return instruction with the initial stack.

```
                     Low Memory
              +-----+-----+
              |     .     |
              | stack area|
              |     .     |
              +-----+-----+
  SS : SP ->  |    IP     |
              +-----+-----+
              |     0     |  (CS)
              +-----+-----+
              |     0     |  (Flags)
              +-----+-----+
```

INT 0, INT 1,
INT 2, INT 3,
INT 4

The initial interrupt vectors for the first five interrupt types can be set by filling in these fields. The first word of each field is the Instruction Pointer (IP) and the second word is the Code Segment (CS) of the interrupt routine which will service these interrupts. Those fields which are zero will be initialized to be the same as the calling processes interrupt vectors. These fields are typically initialized to be 0.

CS,DS,ES,SS

The initial Segment Addresses for the new process are taken from these fields. Those fields that are zero are initialized to be the same as the calling processes Data Segment.

INT 224,
INT 225

Interrupts 224 and 225 are used to communicate with MP/M-86 by typicalprograms. These interrupt vectors will be initialized to be the same as the calling process if these values are zero. The ability to change these values allows a run-time System to intercept MP/M-86 calls that its children make. The suggested protocal is to keep INT 225 pointing to the MP/M-86 entry point and changing INT 224 to point to an internal routine. When a child process does an INT 224, the internal routine can filter calls to MP/M-86 using INT 225 for the actual MP/M-86 call.

RESERVED	These fields are used internally and must be intialized to zero.

```
*************************************************
*                                               *
*  FUNCTION 145:  SET PRIORITY                  *
*                                               *
*************************************************
*                                               *
*    Set the Priority of the Calling Process    *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 091H                       *
*                DL: Priority                   *
*                                               *
*  Return Values:                               *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The SET PRIORITY function sets the priority of the calling process to the specified value. This function is useful in situations where a process needs to have a high priority during an initialization phase, but afterwords can run at a lower priority.

```
**************************************************
*                                                *
*  FUNCTION 146:  ATTACH CONSOLE                 *
*                                                *
**************************************************
*                                                *
*   Attach default console to calling process    *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 092H                        *
*                                                *
*  Return Values:                                *
*                CX: Error Code                  *
*                                                *
**************************************************
```

The ATTACH CONSOLE function attaches the default console to the calling process. If the console is already owned by the calling process or if it is not owned by another process, the ATTACH CONSOLE function will immediately return with ownership established and verified. If another process owns the console, the calling process waits until the console becomes available. When the console becomes free through a DETACH CONSOLE call, the process that is waiting for the console with the highest priority will obtain it. The ATTACH CONSOLE function is called internally by all console I/O functions except the RAW CONSOLE functions.

```
************************************************
*                                              *
*  FUNCTION 147:  DETACH CONSOLE               *
*                                              *
************************************************
*                                              *
*  Detach default console from calling process *
*                                              *
************************************************
*                                              *
*  Entry Parameters:                           *
*      Register  CL: 093H                      *
*                                              *
*  Return Values:                              *
*                CX: Error Code                *
*                                              *
************************************************
```

The DETACH CONSOLE function detaches the default console from the calling process. If the default console is not attached to the calling process, no action is taken. If other processes are waiting to attach to the console, the process with the highest priority will attach the console. If there are more than one process with the same priority waiting for the console, it is given on a first-come first-serve basis.

```
*************************************************
*                                               *
*  FUNCTION 148:  SET CONSOLE                   *
*                                               *
*************************************************
*                                               *
*   Set the calling process's default console   *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 094H                       *
*                DL: Console Number             *
*                                               *
*  Return Values:                               *
*                AX: 0 if successful            *
*                    0ffffH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The SET CONSOLE function changes the calling process's default console to the value specified. If the console number specified is not one supported by this particular implementation of MP/M-86, the function returns an Error Code, and does not change the default console. If the console number is valid, the function detaches the previous default console from the calling process. The SET CONSOLE function then attaches the new console to the calling process through the ATTACH CONSOLE function. If another process already owns the new console, the calling process waits until the console becomes available.

```
*************************************************
*                                               *
*  FUNCTION 149:  ASSIGN CONSOLE                *
*                                               *
*************************************************
*                                               *
*   Assign default console to another process   *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*       Register  CL: 095H                      *
*                 DX: ACB Address - Offset      *
*                 DS: ACB Address - Segment     *
*                                               *
*  Return Values:                               *
*                 CX: Error Code                *
*                                               *
*************************************************
```

```
    +-----+-----+-----+-----+
 00 | CNS |MATCH|    PD     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 04 |                     NAME                      |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 6-16. Assign Control Block (ACB)

CNS Console to assign

MATCH Boolean, if 0FFH, the process being assigned the console must have the CNS as its default console for a successful ASSIGN. IF 0H, no check is made.

PD Process ID of the process being assigned the console. If this field is zero, a search is made of the thread list for a process whose name is NAME. This field must either be zero or a valid Process ID. If this value is not a valid PD, an error occurs.

NAME 8-byte process name to search for. An error occurs if a process by this name does not exist.

The ASSIGN CONSOLE function directly assigns the specified console to a specified process. This function overrides the normal mechanism of the ATTACH and DETACH functions. The function returns an Error Code if a process besides the calling process owns the console. The function ignores other processes waiting to attach to the specified console, and they will continue to wait until the current owner either calls the DETACH function or terminates.

```
*************************************************
*                                               *
*  FUNCTION 150:  COMMAND LINE INTERPRETER      *
*                                               *
*************************************************
*                                               *
*       Interpret and Execute Command Line      *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 096H                       *
*                DX: CLBUF Address - Offset     *
*                DS: CLBUF Address - Segment    *
*                                               *
*  Return Values:                               *
*                AX: 0 if successful            *
*                    0ffffH on error            *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The COMMAND LINE INTERPRETER function obtains an ASCII command from the Command Line Buffer (CLBUF) and then executes it. If the calling process is attached to its default console, the CLI function assigns the console to either the newly created process, or to the Resident System Process (RSP) that will act on the command. The calling process must reattach to its default console before accessing it.

The CLI function calls the PARSE FILENAME function to parse the command line. If an error occurs in the PARSE FILENAME function, the CLI function returns to the calling process with the Error Code set to the same code that the PARSE FILENAME function returned.

If there is no disk specification, for the command, the CLI function will try to open a System Queue with the same name as the command. If the open operation is successful, and the queue is an RSP-type queue, the CLI function looks for a process with the same name and assigns the calling process's default console to the RSP. The CLI function then writes the command tail to the RSP queue. If the queue is full, the function returns an Error Code to the calling process. If for any reason the RSP cannot be found, the CLI assumes the command is on disk and continues.

The CLI function opens a file with the filename being the command and the file type being CMD. If the command has an explicit disk specification, and the OPEN FILE function fails, the CLI function returns an Error Code to the calling process. If there is no disk specification with the command, the CLI function attempts to open the command file on the default system disk. If the OPEN FILE function succeeds, the CLI function checks the file to verify the SYSTEM attribute is on. If this second OPEN FILE function fails or if the DIR attribute is on, the CLI function returns an Error Code to the calling process.

Once the CLI function succeeds in opening the command file, it calls the PROGRAM LOAD function. The PROGRAM LOAD function finds, and then loads the file into an appropriate memory space. If the PROGRAM LOAD function encounters any errors, the CLI function returns to the calling process with the Error Code set by the LOAD function.

A successful load operation establishes the command file in memory with its Base Page partially initialized. The CLI function then continues parsing the command tail to set up the Base Page values from 050h to 0FFh.

The CLI function initializes an unused Process Descriptor from the internal PD Table, a UDA and a 96-byte stack area. The UDA and stack are dynamically allocated from memory. The CLI function then calls the CREATE PROCESS function. If the CLI function encounters an error in any of these steps, it releases all memory segments allocated for the new command, as well as the Process Descriptor, and then returns with the appropriate Error Code set.

Once the CREATE PROCESS function returns successfully, the CLI function assigns the calling process' default console to the new process and then returns.

```
*************************************************
*                                               *
*  FUNCTION 151:  CALL RPL                      *
*                                               *
*************************************************
*                                               *
*               Call a function in a            *
*            Resident Procedure Library         *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 097H                       *
*                DX: CPB Address - Offset       *
*                DS: CPB Address - Segment      *
*                                               *
*  Return Values:                               *
*                AX: 01H if RPL not found       *
*                    RPL return parameter       *
*                BX: same as AX                 *
*                ES: RPL return segment if addr *
*                CX: Error Code                 *
*                                               *
*************************************************
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|                       NAME                    |
+-----+-----+-----+-----+-----+-----+-----+-----+
|   PARAM   |
+-----+-----+
```

Figure 6-17. Call Parameter Block (CPB)

NAME Name of Resident Procedure, eight ASCII characters

PARAM Parameter to send to the Resident Procedure

The CALL RPL function permits a process to call a function in an optional Resident Procedure Library (RPL). Resident Procedure Libraries are optionally included in MP/M-86 by GENSYS at system generation time.

The CALL RPL function opens a System Queue by the name specified. If the OPEN QUEUE function succeeds, Function 151 checks the queue to verify it is an RPL-type queue. If either the OPEN QUEUE call fails or if it is not an RPL-type queue, Function 151 returns to the calling process with an Error Code. The CALL RPL function reads a message from the queue that contains the address of the specified function. It then places the PARAM field of the CPB in register DX, and the calling processes Data Segment address in register DS. The CALL RPL function does a Far Call to the address it obtains from the queue message. Upon return from the RPL, the function copies the BX register to the AX register and then returns to the calling process.

Note: The CALL RPL function does not write the address of the Resident Procedure back to the queue. The Resident Procedure itself must do this. If the Resident Procedure is to be reentrant, it must write the message into the queue upon entry. If it is to be serially reuseable, the procedure must write the message just before returning.

```
*************************************************
*                                               *
*  FUNCTION 152:  PARSE FILENAME                *
*                                               *
*************************************************
*                                               *
*   Parse an ASCII string and initialize a FCB  *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 098H                       *
*                DX: PFCB Address - Offset      *
*                DS: PFCB Address - Segment     *
*                                               *
*  Return Values:                               *
*                AX: 0FFFFH if error            *
*                    0 if next item to parse is *
*                       end of line             *
*                    address of next item to    *
*                       parse                   *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

```
+-----+-----+-----+-----+
| FILENAME  |  FCBADR   |
+-----+-----+-----+-----+
```

Figure 6-18. Parse Filename Control Block (PFCB)

FILENAME Offset of an ASCII file specification to parse. The offset is relative to the same Data Segment as the PFCB.

FCBADR Offset of a File Control Block to initialize. The offset is relative to the same Data Segment as the PFCB

The PARSE FILENAME function parses an ASCII file specification (FILENAME) and prepares a File Control Block (FCB). The calling process passes the address of a data structure called the Parse Filename Control Block, (PFCB) in register DX. The PFCB contains the address of the ASCII filename string followed by the address of the target FCB.

Function 152 assumes the file specification to be in the following form:

{D:}{FILENAME}{.TYP}{;PASSWORD}

where those items enclosed in curly brackets are optional.

The PARSE FILENAME function parses the first file specification it finds in the input string. The function first eliminates leading blanks and tabs. The function then assumes the file specification ends on the first delimiter it hits that is out of context with the specific field it is parsing. For instance, if it finds a colon (:) and it is not the second character of the file specification, the colon delimits the whole file specification. The function recognizes the following characters as delimiters:

space
tab
return
null
; (semicolon) - except before password field
= (equal)
< (less than)
> (greater than)
. (dot) - except after filename and before type
: (colon) - except before filename and after drive
, (comma)
[(left square bracket)
] (right square bracket)
/ (slant)
$ (dollar)

If the function reaches a non-graphic character (in the range 1 through 31), not listed above, it treats it as an error.

The Parse Filename function initializes the specified FCB as follows:

byte 0 — The drive field is set to the specified drive. If the drive is not specified, the default value is used. 0=default, 1=A, 2=B, etc.

byte 1-8 — The name is set to the specified file name. All letters are converted to upper-case. If the name is not eight characters long, the remaining bytes in the filename field are padded with blanks. If the filename has an asterick (*), all remaining bytes in the filename field are filled in with question marks (?). The function returns an error if the filename is more than eight bytes long.

byte 9-11 — The type is set to the specified file type. If no type is specified, the type field is initialized to blanks. All letters are converted to upper-case. If the type is not three characters long, the remaining bytes in the file type field are padded with blanks. If an asterick (*) occurs, all remaining bytes are filled in with question marks

	(?). The function returns an error if the type field is more than 3 bytes long.
byte 12-15	Filled in with zeros
byte 16-23	The password field is set to the specified password. If no password is specified, it is initialized to blanks. If the password is not eight characters long, remaining bytes are padded with blanks. All letters are converted to upper-case. The function returns an error if the password field is more than eight bytes long.
byte 24-25	The offset of the beginning of the password in the FILENAME string is placed here. If no password is specified, this field is set to zero. Note that the password indicated by this field is in the FILENAME string which is not modified by the PARSE FILENAME function. If there are any lower-case characters in the password, they must be converted to upper-case to ensure the password matches the password field of the FCB.
byte 26	The number of characters in the specified password is placed here. If no password is specified, this field is set to zero.

If the function encounters an error, it sets all fields that have not been parsed are set to their default values, and then returns 0FFFFh in register AX indicating the error.

On a successful parse, the PARSE FILENAME function checks the next item in the FILENAME string. It skips over trailing blanks and tabs and look at the next character. If the character is a null (20H) or carriage return (0dH), it returns a 0 indicating the end of the FILENAME string. If the next character is a delimiter, it returns the address of the delimiter. If the next character is not a delimiter, it returns the address of the delimiting blank or tab.

If the first non-blank or non-tab character in the FILENAME string is a null or carriage return, the PARSE FILENAME function returns a 0 indicating the end of string, and initializes the FCB to its default values.

If the PARSE FILENAME function is to be used to parse a subsequent filename in the FILENAME string, the returned address should be advanced over the delimiter before placing it in the PFCB.

```
***********************************************
*                                             *
*  FUNCTION 153:  GET CONSOLE                 *
*                                             *
***********************************************
*                                             *
*  Return the Calling Process' Default Console *
*                                             *
***********************************************
*                                             *
*  Entry Parameters:                          *
*      Register  CL: 099H                     *
*                                             *
*  Return Values:                             *
*                AL: Console number           *
*                BL: Same as AL               *
*                CX: Error Code               *
*                                             *
***********************************************
```

The GET CONSOLE function returns the calling processes default console.

```
*************************************************
*                                               *
*  FUNCTION 154:  GET SYSDAT ADDRESS            *
*                                               *
*************************************************
*                                               *
*   Return the address of the System Data Area  *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 09AH                       *
*                                               *
*  Return Values:                               *
*                AX: SYSDAT Address - Offset    *
*                BX: Same as AX                 *
*                ES: SYSDAT Address - Segment   *
*                                               *
*************************************************
```

The GET SYSDAT function returns the address of the System Data Area. The System Data Area contains all Process Descriptors, Queue Descriptors, the roots of system lists and other internal data that is used by MP/M-86. See the MP/M-86 System Guide for the format of the System Data Area.

```
*************************************************
*                                               *
*  FUNCTION 155:  GET DATE AND TIME             *
*                                               *
*************************************************
*                                               *
*        Get Current System Time and Day        *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 09BH                       *
*                DX: TOD Address - Offset       *
*                DS: TOD Address - Segment      *
*                                               *
*  Return Values:                               *
*                TOD filled in                  *
*                                               *
*************************************************
```

```
+-----+-----+-----+-----+-----+
|    DAY    |HOUR | MIN | SEC |
+-----+-----+-----+-----+-----+
```

Figure 6-19. Time Of Day Structure (TOD)

DAY — The number of days since 1 January 1978. The day is stored as a 16-bit integer.

HOUR — The current hour of the current day. The hour is represented as a 24 hour clock in 2 binary coded decimal (BCD) digits.

MIN — The current minute of the current hour. The minute is stored as 2 BCD digits.

SEC — The current second of the current minute. The second is stored as 2 BCD digits.

The GET DATE AND TIME function returns the current encoded date and time in the TOD structure passed by the calling process.

```
**************************************************
*                                                *
*  FUNCTION 156:  Return PD Address              *
*                                                *
**************************************************
*                                                *
*  Return the Address of the calling process's   *
*              Process Descriptor                *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 09CH                        *
*                                                *
*  Return Values:                                *
*                AX: PD Address - Offset         *
*                BX: Same as AX                  *
*                ES: PD Address - Segment        *
*                                                *
**************************************************
```

The RETURN PROCESS DESCRIPTOR ADDRESS function obtains the address of the calling process' Process Descriptor. The format of the Process descriptor is described in the CREATE PROCESS function description.

```
*************************************************
*                                               *
*  FUNCTION 157:  ABORT SPECIFIED PROCESS       *
*                                               *
*************************************************
*                                               *
*  Terminate a Process by Name or PD Address    *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 09DH                       *
*                DX: APB Address - Offset       *
*                DS: APB Address - Segment      *
*                                               *
*  Return Values:                               *
*                AL: Return Code                *
*                BL: Same as AL                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

```
   +-----+-----+-----+-----+-----+-----+
00 |     PD    |    TERM   | CNS | 00  |
   +-----+-----+-----+-----+-----+-----+-----+-----+
06 |                    NAME                       |
   +-----+-----+-----+-----+-----+-----+-----+-----+
```

Fugure 6-20. Abort Parameter Block (APB)

PD — Process Descriptor Offset of the Process to be terminated. If this field is zero, a match is attempted with the NAME and CNS fields to find the process. If this field is non-zero, the NAME and CNS fields are ignored.

TERM — Termination Code. This field corresponds to the Termination Code of Function 143. If the low-order byte is 0FFH, Function 143 can abort a specified system process; otherwise a system process is not affected. A system process is identified by the SYS flag in the Process Descriptor's FLAG field.

00 — This field is reserved for future use and must be set to zero.

CNS — Default console of Process to be aborted. If the PD field is 0, the ABORT SPECIFIED PROCESS function scans the Thread List for a PD with the same NAME and CNS fields as specified in the APB. Function 157 only aborts the first process that it finds. Subsequent calls must be made to abort all processes with the same NAME and CNS.

NAME Name of the process to be aborted. As in the CNS field, the NAME field is used to find the process to be aborted. This is only used if the PD field is 0.

The ABORT SPECIFIED PROCESS function permits a process to terminate another specified process. The calling process passes the address of a data structure called an Abort Parameter Block, initialized as described above.

If the Process Descriptor address is known, it can be filled in and the process name and console can be omitted. Otherwise, the Process Descriptor address field should be a 0 and the process name and console must be specified. In either case, the calling process must supply the termination code, which is the same parameter passed to the TERMINATE PROCESS function.

```
*************************************************
*                                               *
*  FUNCTION 158:  ATTACH LIST                   *
*                                               *
*************************************************
*                                               *
*       Attach to the Calling Process's         *
*            Default List Device                *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 09EH                       *
*                                               *
*  Return Values:                               *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The ATTACH LIST function attaches the default list device of the calling process. If the list device is already attached to some other process, the calling process relinquishes the CPU until the other process detaches from the list device. When the list device becomes free and the calling process is the highest priority process waiting for the list device, the attach operation takes place.

```
*************************************************
*                                               *
*  FUNCTION 159:  DETACH LIST                   *
*                                               *
*************************************************
*                                               *
*       Detach the Calling Process's            *
*           Default List Device                 *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 09FH                       *
*                                               *
*  Return Values:                               *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The DETACH LIST function detaches the default list device of the calling process. If the list device is not currently attached, no action takes place.

```
*************************************************
*                                               *
*  FUNCTION 160:  SET LIST                      *
*                                               *
*************************************************
*                                               *
* Set the Calling Process's Default List Device *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*       Register  CL: 0A0H                      *
*                 DL: List Device               *
*                                               *
*  Return Values:                               *
*                 CX: Error Code                *
*                                               *
*************************************************
```

The SET LIST function detaches the list device currently attached to the calling process and then attaches the specified list device. If the list device to be attached is already attached to another process, the calling process relinquishes the CPU until the other process detaches from the list device. When the list device becomes free and the calling process is the highest priority process waiting for the device, the attach operation takes place.

```
*************************************************
*                                               *
*  FUNCTION 161:  CONDITIONAL ATTACH LIST       *
*                                               *
*************************************************
*                                               *
*        Conditionally Attach to the            *
*           Default List Device                 *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 0A1H                       *
*                                               *
*  Return Values:                               *
*                AX: 0 if attach 'OK'           *
*                    0FFFFH on failure          *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The CONDITIONAL ATTACH LIST function attachs the default list device of the calling process only if the list device is currently available.

If the list device is currently attached to another process, the function returns a value of 0FFH indicating that the list device could not be attached. The function returns a value of 0 to indicate that either the list device is already attached to the process, or that it was unattached and a successful attach operation was made.

```
**********************************************
*                                            *
*  FUNCTION 162:  CONDITIONAL ATTACH CONSOLE *
*                                            *
**********************************************
*                                            *
*  Conditionally Attach to the Default Console *
*                                            *
**********************************************
*                                            *
*  Entry Parameters:                         *
*      Register  CL: 0A2H                    *
*                                            *
*  Return Values:                            *
*                AX: 0 if attach 'OK'        *
*                    0FFFFH on failure       *
*                BX: Same as AX              *
*                CX: Error Code              *
*                                            *
**********************************************
```

The CONDITIONAL ATTACH CONSOLE function attaches the default console of the calling process only if the console is currently unattached.

If the console is currently attached to another process, the function returns a value of 0FFH indicating that the console could not be attached. The function returns a value of 0 to indicate that either the console is already attached to the process or that it was unattached and a successful attach operation was made.

```
*************************************************
*                                               *
*  FUNCTION 163:  RETURN MP/M VERSION NUMBER    *
*                                               *
*************************************************
*                                               *
*  Return the version of current MP/M-86 system *
*                                               *
*************************************************
*                                               *
*  Entry Parameters:                            *
*      Register  CL: 0A3H                       *
*                                               *
*  Return Values:                               *
*                AX: Version Number (01120H)    *
*                BX: Same as AX                 *
*                CX: Error Code                 *
*                                               *
*************************************************
```

The RETURN MP/M VERSION NUMBER function provides information which allows version independent programming. The function returns a two-byte value, with AH set to 011H for MP/M-86 and AL set to the MP/M-86 verion level. A value of 01120H indicates MP/M-86 2.0.

```
**************************************************
*                                                *
*  FUNCTION 164:  GET LIST NUMBER                *
*                                                *
**************************************************
*                                                *
*        Return the Calling Process's            *
*             Default List Device                *
*                                                *
**************************************************
*                                                *
*  Entry Parameters:                             *
*      Register  CL: 0A4H                        *
*                                                *
*  Return Values:                                *
*                AL: List Device Number          *
*                BL: Same as AL                  *
*                CX: Error Code                  *
*                                                *
**************************************************
```

The GET LIST NUMBER function returns the default list device number of the calling process.

SECTION 7

INTRODUCTION TO ASM

7.1 Assembler Operation

ASM-86 processes an 8086 assembly language source file in three passes and produces three output files, including an 8086 machine language file in hexadecimal format. This object file may be in either Intel or Digital Research hex format, which are described in Appendix C. ASM-86 is shipped in two forms: an 8086 cross-assembler designed to run under CP/M on an Intel 8080 or Zilog Z-80 based system, and a 8086 assembler designed to run under MP/M-86 on an Intel 8086 or 8088 based system. ASM-86 typically produces three output files from one input file as shown in Figure 7-1, below.

```
                                                  ===============
                                      --------> | LIST FILE   |
                                      |           ===============
                    ============      |
==========          |          |      |           ===============
| SOURCE |------->|  ASM-86  |----------------> | HEX FILE    |
==========          |          |      |           ===============
                    ============      |
                                      |           ===============
                                      --------> | SYMBOL FILE |
                                                  ===============
```

<file name>.A86	-	contains source
<file name>.LST	-	contains listing
<file name>.H86	-	contains assembled program in hexadecimal format
<file name>.SYM	-	contains all user-defined symbols

Figure 7-1. ASM-86 Source and Object Files

Figure 7-1 also lists ASM-86 filename extensions. ASM-86 accepts a source file with any three letter extension, but if the extension is omitted from the invoking command, it looks for the specified file name with the extension .A86 in the directory. If the file has an extension other than .A86 or has no extension at all, ASM-86 returns an error message.

The other extensions listed in Figure 7-1 identify ASM-86 output files. The .LST file contains the assembly language listing with any error messages. The .H86 file contains the machine language program in either Digital Research or Intel hexadecimal format. The .SYM file lists any user-defined symbols.

Invoke ASM-86 by entering a command of the following form:

ASM86 <source filename> [$ <optional parameters>]

Section 7.2 explains the optional parameters. Specify the source file in the following form:

[<optional drive>:]<filename>[.<optional extension>]

where

<optional drive>	is a valid drive letter specifying the source file's location. Not needed if source is on current drive.
<filename>	is a valid CP/M filename of 7 to 8 characters.
<optional extension>	is a valid file extension of 1 to 3 characters, usually .A86.

Some examples of valid ASM-86 commands are:

A>**ASM86 B:BIOS88**

A>**ASM86 BIOS88.A86 $FI AA HB PB SB**

A>**ASM86 D:TEST**

Once invoked, ASM-86 responds with the message:

CP/M 8086 ASSEMBLER VER x.x

where x.x is the ASM-86 version number. ASM-86 then attempts to open the source file. If the file does not exist on the designated drive, or does not have the correct extension as described above, the assembler displays the message:

NO FILE

If an invalid parameter is given in the optional parameter list, ASM-86 displays the message:

PARAMETER ERROR

After opening the source, the assembler creates the output files. Usually these are placed on the current disk drive, but they may be redirected by optional parameters, or by a drive specification in the the source file name. In the latter case, ASM-86 directs the output files to the drive specified in the source file name.

During assembly, ASM-86 aborts if an error condition such as disk full or symbol table overflow is detected. When ASM-86 detects an error in the source file, it places an error message line in the listing file in front of the line containing the error. Each error message has a number and gives a brief explanation of the error. Appendix H lists ASM-86 error messages. When the assembly is complete, ASM-86 displays the message:

```
END OF ASSEMBLY. NUMBER OF ERRORS: n
```

7.2 Optional Run-time Parameters

The dollar-sign character, $, flags an optional string of run-time parameters. A parameter is a single letter followed by a single letter device name specification. The parameters are shown in Table 7-1, below.

Table 7-1. Run-time Parameter Summary

Parameter	To Specify	Valid Arguments
A	source file device	A, B, C, ... P
H	hex output file device	A ... P, X, Y, Z
P	list file device	A ... P, X, Y, Z
S	symbol file device	A ... P, X, Y, Z
F	format of hex output file	I, D

All parameters are optional, and can be entered in the command line in any order. Enter the dollar sign only once at the beginning of the parameter string. Spaces may separate parameters, but are not required. No space is permitted, however, between a parameter and its device name.

A device name must follow parameters A, H, P and S. The devices are labeled:

```
A, B, C, ... P or X, Y, Z
```

Device names A through P respectively specify disk drives A through P. X specifies the user console (CON:), Y specifies the line printer (LST:), and Z suppresses output (NUL:).

If output is directed to the console, it may be temporarily stopped at any time by typing a control-S. Restart the output by typing a second control-S or any other character.

The F parameter requires either an I or a D argument. When I is specified, ASM-86 produces an object file in Intel hex format. A D argument requests Digital Research hex format. Appendix C discusses these formats in detail. If the F parameter is not entered in the command line, ASM-86 produces Digital Research hex format.

Table 7-2. Run-time Parameter Examples

Command Line	Result
ASM86 IO	Assemble file IO.A86, produce IO.HEX, IO.LST and IO.SYM, all on the default drive.
ASM86 IO.ASM $ AD SZ	Assemble file IO.ASM on device D, produce IO.LST and IO.HEX, no symbol file.
ASM86 IO $ PY SX	Assemble file IO.A86, produce IO.HEX, route listing directly to printer, output symbols on console.
ASM86 IO $ FD	Produce Digital Research hex format.
ASM86 IO $ FI	Produce Intel hex format.

7.3 Aborting ASM-86

You may abort ASM-86 execution at any time by hitting any key on the console keyboard. When a key is pressed, ASM-86 responds with the question:

```
USER BREAK. OK(Y/N)?
```

A Y response aborts the assembly and returns to the operating system. An N response continues the assembly.

SECTION 8

ELEMENTS OF ASM-86 ASSEMBLY LANGUAGE

8.1 ASM-86 Character Set

ASM-86 recognizes a subset of the ASCII character set. The valid characters are the alphanumerics, special characters, and non-printing characters shown below:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9

+ - * / = ( ) [ ] ; ' . ! , _ : @ $

space, tab, carriage-return, and line-feed

Lower-case letters are treated as upper-case except within strings. Only alphanumerics, special characters, and spaces may appear within a string.

8.2 Tokens and Separators

A token is the smallest meaningful unit of an ASM-86 source program, much as a word is the smallest meaningful unit of an English composition. Adjacent tokens are commonly separated by a blank character or space. Any sequence of spaces may appear wherever a single space is allowed. ASM-86 recognizes horizontal tabs as separators and interprets them as spaces. Tabs are expanded to spaces in the list file. The tab stops are at each eighth column.

8.3 Delimiters

Delimiters mark the end of a token and add special meaning to the instruction, as opposed to separators, which merely mark the end of a token. When a delimiter is present, separators need not be used. However, separators after delimiters can make your program easier to read.

Table 8-1 describes ASM-86 separators and delimiters. Some delimiters are also operators and are explained in greater detail in Section 8.6.

Table 8-1. Separators and Delimiters

Character	Name	Use
20H	space	separator
09H	tab	legal in source files, expanded in list files
CR	carriage return	terminate source lines
LF	line feed	legal after CR; if within source lines, it is interpreted as a space
;	semicolon	start comment field
:	colon	identifies a label, used in segment override specification
.	period	forms variables from numbers
$	dollar sign	notation for "present value of location pointer"
+	plus	arithmetic operator for addition
-	minus	arithmetic operator for subtraction
*	asterisk	arithmetic operator for multiplication
/	slash	arithmetic operator for division
@	at-sign	legal in identifiers
_	underscore	legal in identifiers
!	exclamation point	logically terminates a statement, thus allowing multiple statements on a single source line
'	apostrophe	delimits string constants

8.4 Constants

A constant is a value known at assembly time that does not change while the assembled program is executed. A constant may be either an integer or a character string.

8.4.1 Numeric Constants

A numeric constant is a 16-bit value in one of several bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. The radix indicators are shown in Table 8-2, below.

Table 8-2. Radix Indicators for Constants

Indicator	Constant Type	Base
B	binary	2
O	octal	8
Q	octal	8
D	decimal	10
H	hexadecimal	16

ASM-86 assumes that any numeric constant not terminated with a radix indicator is a decimal constant. Radix indicators may be upper or lower case.

A constant is thus a sequence of digits followed by an optional radix indicator, where the digits are in the range for the radix. Binary constants must be composed of 0's and 1's. Octal digits range from 0 to 7; decimal digits range from 0 to 9. Hexadecimal constants contain decimal digits as well as the hexadecimal digits A (10D), B (11D), C (12D), D (13D), E (14D), and F (15D). Note that the leading character of a hexadecimal constant must be either a decimal digit so that ASM-86 cannot confuse a hex constant with an identifier, or leading 0 to prevent this problem. The following are valid numeric constants:

```
1234     1234D    1100B    1111000011110000B
1234H    0FFEH    33770    13772Q
33770    0FE3H    1234d    0ffffh
```

8.4.2 Character Strings

ASM-86 treats an ASCII character string delimited by apostrophes as a string constant. All instructions accept only one- or two-character constants as valid arguments. Instructions treat a one-character string as an 8-bit number. A two-character string is treated as a 16-bit number with the value of the second character in the low-order byte, and the value of the first character in the high-order byte.

The numeric value of a character is its ASCII code. ASM-86 does not translate case within character strings, so both upper- and lower-case letters can be used. Note that only alphanumerics, special characters, and spaces are allowed within strings.

A DB assembler directive is the only ASM-86 statement that may contain strings longer than two characters. The string may not exceed 255 bytes. Include any apostrophe to be printed within the string by entering it twice. ASM-86 interprets the two keystrokes '' as a single apostrophe. Table 8-3 shows valid strings and how they appear after processing:

Table 8-3. String Constant Examples

```
               'a' -> a
          'Ab''Cd' -> Ab'Cd
     'I like CP/M' -> I like CP/M
              '''' -> '
'ONLY UPPER CASE' -> ONLY UPPER CASE
'only lower case' -> only lower case
```

8.5 Identifiers

Identifiers are character sequences which have a special, symbolic meaning to the assembler. All identifiers in ASM-86 must obey the following rules:

1. The first character must be alphabetic (A,...Z, a,...z).

2. Any subsequent characters can be either alphabetical or a numeral (0,1,.....9). ASM-86 ignores the special characters @ and _, but they are still legal. For example, a_b becomes ab.

3. Identifiers may be of any length up to the limit of the physical line.

Identifiers are of two types. The first are keywords, which have predefined meanings to the assembler. The second are symbols, which are defined by the user. The following are all valid identifiers:

```
NOLIST
WORD
AH
Third_street
How_are_you_today
variable@number@1234567890
```

8.5.1 Keywords

A keyword is an identifier that has a predefined meaning to the assembler. Keywords are reserved; the user cannot define an identifier identical to a keyword. For a complete list of keywords, see Appendix D.

ASM-86 recognizes five types of keywords: instructions, directives, operators, registers and predefined numbers. 8086 instruction mnemonic keywords and the actions they initiate are defined in Section 10. Directives are discussed in Section 9. Section 8.6 defines operators. Table 8-4 lists the ASM-86 keywords that identify 8086 registers.

Three keywords are predefined numbers: BYTE, WORD, and DWORD. The values of these numbers are 1, 2 and 4, respectively. In addition, a Type attribute is associated with each of these numbers. The keyword's Type attribute is equal to the keyword's numeric value. See Section 8.5.2 for a complete discussion of Type attributes.

Table 8-4. Register Keywords

Register Symbol	Size	Numeric Value	Meaning
AH	1 byte	100 B	Accumulator-High-Byte
BH	1 "	111 B	Base-Register-High-Byte
CH	1 "	101 B	Count-Register-High-Byte
DH	1 "	110 B	Data-Register-High-Byte
AL	1 "	000 B	Accumulator-Low-Byte
BL	1 "	011 B	Base-Register-Low-Byte
CL	1 "	001 B	Count-Register-Low-Byte
DL	1 "	010 B	Data-Register-Low-Byte
AX	2 bytes	000 B	Accumulator (full word)
BX	2 "	011 B	Base-Register "
CX	2 "	001 B	Count-Register "
DX	2 "	010 B	Data-Register "
BP	2 "	101 B	Base Pointer
SP	2 "	100 B	Stack Pointer
SI	2 "	110 B	Source Index
DI	2 "	111 B	Destination Index
CS	2 "	01 B	Code-Segment-Register
DS	2 "	11 B	Data-Segment-Register
SS	2 "	10 B	Stack-Segment-Register
ES	2 "	00 B	Extra-Segment-Register

8.5.2 Symbols and Their Attributes

A symbol is a user-defined identifier that has attributes which specify what kind of information the symbol represents. Symbols fall into three categories:

- variables
- labels
- numbers

<u>Variables</u> identify data stored at a particular location in memory. All variables have the following three attributes:

- Segment - tells which segment was being assembled when the variable was defined.

- Offset - tells how many bytes there are between the beginning of the segment and the location of this variable.

- Type - tells how many bytes of data are manipulated when this variable is referenced.

A Segment may be a code-segment, a data-segment, a stack-segment or an extra-segment depending on its contents and the register that contains its starting address (see Section 9.2). A segment may start at any address divisible by 16. ASM-86 uses this boundary value as the Segment portion of the variable's definition.

The Offset of a variable may be any number between 0 and 0FFFFH or 65535D. A variable must have one of the following Type attributes:

- BYTE
- WORD
- DWORD

BYTE specifies a one-byte variable, WORD a two-byte variable and DWORD a four-byte variable. The DB, DW, and DD directives respectively define variables as these three types (see Section 9). For example, a variable is defined when it appears as the name for a storage directive:

```
VARIABLE  DB 0
```

A variable may also be defined as the name for an EQU directive referencing another label, as shown below:

```
VARIABLE  EQU  ANOTHER_VARIABLE
```

Labels identify locations in memory that contain instruction statements. They are referenced with jumps or calls. All labels have two attributes:

- Segment
- Offset

Label segment and offset attributes are essentially the same as variable segment and offset attributes. Generally, a label is defined when it precedes an instruction. A colon, :, separates the label from instruction; for example:

```
LABEL:   ADD  AX,BX
```

A label may also appear as the name for an EQU directive referencing another label; for example:

```
LABEL   EQU   ANOTHER_LABEL
```

Numbers may also be defined as symbols. A number symbol is treated as if you had explicitly coded the number it represents. For example:

```
Number_five    EQU   5
MOV   AL,Number_five
```

is equivalent to:

```
MOV   AL,5
```

Section 8.6 describes operators and their effects on numbers and number symbols.

8.6 Operators

ASM-86 operators fall into the following categories: arithmetic, logical, and relational operators, segment override, variable manipulators and creators. Table 8-5 defines ASM-86 operators. In this table, a and b represent two elements of the expression. The validity column defines the type of operands the operator can manipulate, using the or bar character, |, to separate alternatives.

Table 8-5. ASM-86 Operators

Syntax	Result	Validity
	Logical Operators	
a XOR b	bit-by-bit logical EXCLUSIVE OR of a and b.	a, b = number
a OR b	bit-by-bit logical OR of a and b.	a, b = number
a AND b	bit-by-bit logical AND of a and b.	a, b = number
NOT a	logical inverse of a: all 0's become 1's, 1's become 0's.	a = 16-bit number

Table 8-5. (continued)

Syntax	Result	Validity
	Relational Operators	
a EQ b	returns 0FFFFH if a = b, otherwise 0.	a, b = unsigned number
a LT b	returns 0FFFFH if a < b, otherwise 0.	a, b = unsigned number
a LE b	returns 0FFFFH if a <= b, otherwise 0.	a, b = unsigned number
a GT b	returns 0FFFFH if a > b, otherwise 0.	a, b = unsigned number
a GE b	returns 0FFFFH if a >= b otherwise 0.	a, b = unsigned number
a NE b	returns 0FFFFH if a <> b, otherwise 0.	a, b = unsigned number
	Arithmetic Operators	
a + b	arithmetic sum of a and b.	a = variable, label or number b = number
a - b	arithmetic difference of a and b.	a = variable, label or number b = number
a * b	does unsigned multiplication of a and b.	a, b = number
a / b	does unsigned division of a and b.	a, b = number
a MOD b	returns remainder of a / b.	a, b = number
a SHL b	returns the value which results from shifting a to left by an amount b.	a, b = number
a SHR b	returns the value which results from shifting a to the right by an amount b.	a, b = number
+ a	gives a.	a = number
- a	gives 0 - a.	a = number

Table 8-5. (continued)

Syntax	Result	Validity
	Segment Override	
<seg reg>: <addr exp>	overrides assembler's choice of segment register.	<seg reg> = CS, DS, SS or ES
	Variable Manipulators, Creators	
SEG a	creates a number whose value is the segment value of the variable or label a.	a = label | variable
OFFSET a	creates a number whose value is the offset value of the variable or label a.	a = label | variable
TYPE a	creates a number. If the variable a is of type BYTE, WORD or DWORD, the value of the number will be 1, 2 or 4, respectively.	a = label | variable
LENGTH a	creates a number whose value is the LENGTH attribute of the variable a. The length attribute is the number of bytes associated with the variable.	a = label | variable
LAST a	if LENGTH a > 0, then LAST a = LENGTH a - 1; if LENGTH a = 0, then LAST a = 0.	a = label | variable
a PTR b	creates virtual variable or label with type of a and attributes of b	a = BYTE | WORD, | DWORD b = <addr exp>
.a	creates variable with an offset attribute of a. Segment attribute is current segment.	a = number
$	creates label with offset equal to current value of location counter; segment attribute is current segment.	no argument

8.6.1 Operator Examples

Logical operators accept only numbers as operands. They perform the boolean logic operations AND, OR, XOR, and NOT. For example:

```
   00FC          MASK    EQU   0FCH
   0080          SIGNBIT EQU   80H
0000 B180                MOV   CL,MASK AND SIGNBIT
0002 B003                MOV   AL,NOT MASK
```

Relational operators treat all operands as unsigned numbers. The relational operators are EQ (equal), LT (less than), LE (less than or equal), GT (greater than), GE (greater than or equal), and NE (not equal). Each operator compares two operands and returns all ones (0FFFFH) if the specified relation is true and all zeros if it is not. For example:

```
   000A          LIMIT1  EQU   10
   0019          LIMIT2  EQU   25
                          .
                          .
                          .
0004 B8FFFF              MOV   AX,LIMIT1 LT LIMIT2
0007 B80000              MOV   AX,LIMIT1 GT LIMIT2
```

Addition and subtraction operators compute the arithmetic sum and difference of two operands. The first operand may be a variable, label, or number, but the second operand must be a number. When a number is added to a variable or label, the result is a variable or label whose offset is the numeric value of the second operand plus the offset of the first operand. Subtraction from a variable or label returns a variable or label whose offset is that of first operand decremented by the number specified in the second operand. For example:

```
   0002          COUNT   EQU   2
   0005          DISP1   EQU   5
000A FF          FLAG    DB    0FFH
                          .
                          .
                          .
000B 2EA00B00            MOV   AL,FLAG+1
000F 2E8A0E0F00          MOV   CL,FLAG+DISP1
0014 B303                MOV   BL,DISP1-COUNT
```

The multiplication and division operators *, /, MOD, SHL, and SHR accept only numbers as operands. * and / treat all operators as unsigned numbers. For example:

```
0016 BE5500              MOV   SI,256/3
0019 B310                MOV   BL,64/4
  0050           BUFFERSIZE    EQU   80
001B B8A000              MOV   AX,BUFFERSIZE * 2
```

Unary operators accept both signed and unsigned operators as shown below:

```
0O1E B123                  MOV     CL,+35
0020 B007                  MOV     AL,2--5
0022 B2F4                  MOV     DL,-12
```

When manipulating variables, the assembler decides which segment register to use. You may override the assembler's choice by specifying a different register with the segment override operator. The syntax for the override operator is <segment register> : <address expression> where the <segment register> is CS, DS, SS, or ES. For example:

```
0024 368B472D              MOV     AX,SS:WORDBUFFER[BX]
0028 268B0E5B00            MOV     CX,ES:ARRAY
```

A variable manipulator creates a number equal to one attribute of its variable operand. SEG extracts the variable's segment value, OFFSET its offset value, TYPE its type value (1, 2, or 4), and LENGTH the number of bytes associated with the variable. LAST compares the variable's LENGTH with 0 and if greater, then decrements LENGTH by one. If LENGTH equals 0, LAST leaves it unchanged. Variable manipulators accept only variables as operators. For example:

```
002D 000000000000 WORDBUFFER      DW      0,0,0
0033 0102030405   BUFFER          DB      1,2,3,4,5
                                  .
                                  .
                                  .
0038 B80500                  MOV     AX,LENGTH BUFFER
003B B80400                  MOV     AX,LAST BUFFER
003E B80100                  MOV     AX,TYPE BUFFER
0041 B80200                  MOV     AX,TYPE WORDBUFFER
```

The PTR operator creates a virtual variable or label, one valid only during the execution of the instruction. It makes no changes to either of its operands. The temporary symbol has the same Type attribute as the left operator, and all other attributes of the right operator as shown below.

```
0044 C60705                  MOV     BYTE PTR [BX], 5
0047 8A07                    MOV     AL,BYTE PTR [BX]
0049 FF04                    INC     WORD PTR [SI]
```

The Period operator, ., creates a variable in the current data segment. The new variable has a segment attribute equal to the current data segment and an offset attribute equal to its operand. Its operand must be a number. For example:

```
004B A10000                  MOV     AX, .0
004E 268B1E0040              MOV     BX, ES: .4000H
```

The Dollar-sign operator, $, creates a label with an offset attribute equal to the current value of the location counter. The label's segment value is the same as the current code segment. This operator takes no operand. For example:

```
0053 E9FDFF          JMP     $
0056 EBFE            JMPS    $
0058 E9FD2F          JMP     $+3000H
```

8.6.2 Operator Precedence

Expressions combine variables, labels or numbers with operators. ASM-86 allows several kinds of expressions which are discussed in Section 8.7. This section defines the order in which operations are executed should more than one operator appear in an expression.

In general, ASM-86 evaluates expressions left to right, but operators with higher precedence are evaluated before operators with lower precedence. When two operators have equal precedence, the left-most is evaluated first. Table 8-6 presents ASM-86 operators in order of increasing precedence.

Parentheses can override normal rules of precedence. The part of an expression enclosed in parentheses is evaluated first. If parentheses are nested, the innermost expressions are evaluated first. Only five levels of nested parentheses are legal. For example:

```
15/3 + 18/9 = 5 + 2 = 7
15/(3 + 18/9) = 15/(3 + 2) = 15/5 = 3
```

Table 8-6. Precedence of Operations in ASM-86

Order	Operator Type	Operators
1	Logical	XOR, OR
2	Logical	AND
3	Logical	NOT
4	Relational	EQ, LT, LE, GT, GE, NE
5	Addition/subtraction	+, -
6	Multiplication/division	*, /, MOD, SHL, SHR
7	Unary	+, -
8	Segment override	<segment override>:
9	Variable manipulators, creators	SEG, OFFSET, PTR, TYPE, LENGTH, LAST
10	Parentheses/brackets	( ), [ ]
11	Period and Dollar	., $

8.7 Expressions

ASM-86 allows address, numeric, and bracketed expressions. An address expression evaluates to a memory address and has three components:

- A segment value
- An offset value
- A type

Both variables and labels are address expressions. An address expression is not a number, but its components are. Numbers may be combined with operators such as PTR to make an address expression.

A numeric expression evaluates to a number. It does not contain any variables or labels, only numbers and operands.

Bracketed expressions specify base- and index- addressing modes. The base registers are BX and BP, and the index registers are DI and SI. A bracketed expression may consist of a base register, an index register, or a base register and an index register.

Use the + operator between a base register and an index register to specify both base- and index-register addressing. For example:

```
MOV  variable[bx],0
MOV  AX,[BX+DI]
MOV  AX,[SI]
```

8.8 Statements

Just as "tokens" in this assembly language correspond to words in English, so are statements analogous to sentences. A statement tells ASM-86 what action to perform. Statements are of two types: instructions and directives. Instructions are translated by the assembler into 8086 machine language instructions. Directives are not translated into machine code but instead direct the assembler to perform certain clerical functions.

Terminate each assembly language statement with a carriage return (CR) and line feed (LF), or with an exclamation point, !, which ASM-86 treats as an end-of-line. Multiple assembly language statements can be written on the same physical line if separated by exclamation points.

The ASM-86 instruction set is defined in Section 10. The syntax for an instruction statement is:

```
[label:]  [prefix]  mnemonic [ operand(s)]  [;comment]
```

where the fields are defined as:

label:

A symbol followed by ":" defines a label at the current value of the location counter in the current segment. This field is optional.

prefix

Certain machine instructions such as LOCK and REP may prefix other instructions. This field is optional.

mnemonic

A symbol defined as a machine instruction, either by the assembler or by an EQU directive. This field is optional unless preceded by a prefix instruction. If it is omitted, no operands may be present, although the other fields may appear. ASM-86 mnemonics are defined in Section 10.

operand(s)

An instruction mnemonic may require other symbols to represent operands to the instruction. Instructions may have zero, one or two operands.

comment

Any semicolon (;) appearing outside a character string begins a comment, which is ended by a carriage return. Comments improve the readability of programs. This field is optional.

ASM-86 directives are described in Section 9. The syntax for a directive statement is:

```
[name] directive operand(s) [;comment]
```

where the fields are defined as:

name

Unlike the label field of an instruction, the name field of a directive is never terminated with a colon. Directive names are legal for only DB, DW, DD, RS and EQU. For DB, DW, DD and RS the name is optional; for EQU it is required.

directive

One of the directive keywords defined in Section 9.

operand(s)

Analogous to the operands to the instruction mnemonics. Some directives, such as DB, DW, and DD, allow any operand while others have special requirements.

comment

Exactly as defined for instruction statements.

SECTION 9

ASSEMBLER DIRECTIVES

9.1 Introduction

Directive statements cause ASM-86 to perform housekeeping functions such as assigning portions of code to logical segments, requesting conditional assembly, defining data items, and specifying listing file format. General syntax for directive statements appears in Section 8.8.

In the sections that follow, the specific syntax for each directive statement is given under the heading and before the explanation. These syntax lines use special symbols to represent possible arguments and other alternatives. Square brackets, [], enclose optional arguments. Angle brackets, <>, enclose descriptions of user-supplied arguments. Do not include these symbols when coding a directive.

9.2 Segment Start Directives

At run-time, every 8086 memory reference must have a 16-bit segment base value and a 16-bit offset value. These are combined to produce the 20-bit effective address needed by the CPU to physically address the location. The 16-bit segment base value or boundary is contained in one of the segment registers CS, DS, SS, or ES. The offset value gives the offset of the memory reference from the segment boundary. A 16-byte physical segment is the smallest relocatable unit of memory.

ASM-86 predefines four logical segments: the Code Segment, Data Segment, Stack Segment, and Extra Segment, which are respectively addressed by the CS, DS, SS, and ES registers. Future versions of ASM-86 will support additional segments such as multiple data or code segments. All ASM-86 statements must be assigned to one of the four currently supported segments so that they can be referenced by the CPU. A segment directive statement, CSEG, DSEG, SSEG, or ESEG, specifies that the statements following it belong to a specific segment. The statements are then addressed by the corresponding segment register. ASM-86 assigns statements to the specified segment until it encounters another segment directive.

Instruction statements must be assigned to the Code Segment. Directive statements may be assigned to any segment. ASM-86 uses these assignments to change from one segment register to another. For example, when an instruction accesses a memory variable, ASM-86 must know which segment contains the variable so it can generate a segment override prefix byte if necessary.

9.2.1 The CSEG Directive

```
CSEG    <numeric expression>
CSEG
CSEG    $
```

This directive tells the assembler that the following statements belong in the Code Segment. All instruction statements must be assigned to the Code Segment. All directive statements are legal within the Code Segment.

Use the first form when the location of the segment is known at assembly time; the code generated is not relocatable. Use the second form when the segment location is not known at assembly time; the code generated is relocatable. Use the third form to continue the Code Segment after it has been interrupted by a DSEG, SSEG, or ESEG directive. The continuing Code Segment starts with the same attributes, such as location and instruction pointer, as the previous Code Segment.

9.2.2 The DSEG Directive

```
DSEG    <numeric expression>
DSEG
DSEG    $
```

This directive specifies that the following statements belong to the Data Segment. The Data Segment primarily contains the data allocation directives DB, DW, DD and RS, but all other directive statements are also legal. Instruction statements are illegal in the Data Segment.

Use the first form when the location of the segment is known at assembly time; the code generated is not relocatable. Use the second form when the segment location is not known at assembly time; the code generated is relocatable. Use the third form to continue the Data Segment after it has been interrupted by a CSEG, SSEG, or ESEG directive. The continuing Data Segment starts with the same attributes as the previous Data Segment.

9.2.3 The SSEG Directive

```
SSEG    <numeric expression>
SSEG
SSEG    $
```

The SSEG directive indicates the beginning of source lines for the Stack Segment. Use the Stack Segment for all stack operations. All directive statements are legal in the Stack Segment, but instruction statements are illegal.

Use the first form when the location of the segment is known at assembly time; the code generated is not relocatable. Use the second form when the segment location is not known at assembly time; the code generated is relocatable. Use the third form to continue the Stack Segment after it has been interrupted by a CSEG, DSEG, or ESEG directive. The continuing Stack Segment starts with the same attributes as the previous Stack Segment.

9.2.4 The ESEG Directive

```
ESEG    <numeric expression>
ESEG
ESEG    $
```

This directive initiates the Extra Segment. Instruction statements are not legal in this segment, but all directive statements are.

Use the first form when the location of the segment is known at assembly time; the code generated is not relocatable. Use the second form when the segment location is not known at assembly time; the code generated is relocatable. Use the third form to continue the Extra Segment after it has been interrupted by a DSEG, SSEG, or CSEG directive. The continuing Extra Segment starts with the same attributes as the previous Extra Segment.

9.3 The ORG Directive

```
ORG     <numeric expression>
```

The ORG directive sets the offset of the location counter in the current segment to the value specified in the numeric expression. Define all elements of the expression before the ORG directive because forward references may be ambiguous.

In most segments, an ORG directive is unnecessary. If no ORG is included before the first instruction or data byte in a segment, assembly begins at location zero relative to the beginning of the segment. A segment can have any number of ORG directives.

9.4 The IF and ENDIF Directives

```
        IF      <numeric expression>
                < source line 1 >
                < source line 2 >
                        .
                        .
                        .
                < source line n >
                ENDIF
```

The IF and ENDIF directives allow a group of source lines to be included or excluded from the assembly. Use conditional directives to assemble several different versions of a single source program.

When the assembler finds an IF directive, it evaluates the numeric expression following the IF keyword. If the expression evaluates to a non-zero value, then <source line 1> through <source line n> are assembled. If the expression evaluates to zero, then all lines are listed but not assembled. All elements in the numeric expression must be defined before they appear in the IF directive. Nested IF directives are not legal.

9.5 The INCLUDE Directive

```
        INCLUDE    <file name>
```

This directive includes another ASM-86 file in the source text. For example:

```
        INCLUDE   EQUALS.A86
```

Use INCLUDE when the source program resides in several different files. INCLUDE directives may not be nested; a source file called by an INCLUDE directive may not contain another INCLUDE statement. If <file name> does not contain a file type, the file type is assumed to be .A86. If no drive name is specified with <file name>, ASM-86 assumes the drive containing the source file.

9.6 The END Directive

```
        END
```

An END directive marks the end of a source file. Any subsequent lines are ignored by the assembler. END is optional. If not present, ASM-86 processes the source until it finds an End-Of-File character (1AH).

9.7 The EQU Directive

```
symbol  EQU  <numeric expression>
symbol  EQU  <address expression>
symbol  EQU  <register>
symbol  EQU  <instruction mnemonic>
```

The EQU (equate) directive assigns values and attributes to user-defined symbols. The required symbol name may not be terminated with a colon. The symbol cannot be redefined by a subsequent EQU or another directive. Any elements used in numeric or address expressions must be defined before the EQU directive appears.

The first form assigns a numeric value to the symbol, the second a memory address. The third form assigns a new name to an 8086 register. The fourth form defines a new instruction (sub)set. The following are examples of these four forms:

```
     0005             FIVE    EQU    2*2+1
     0033             NEXT    EQU    BUFFER
     0001             COUNTER EQU    CX
                      MOVVV   EQU    MOV
                                     .
                                     .
                                     .
005D 8BC3                     MOVVV  AX,BX
```

9.8 The DB Directive

```
[symbol] DB <numeric expression>[,<numeric expression>..]
[symbol] DB <string constant>[,<string constant>...]
```

The DB directive defines initialized storage areas in byte format. Numeric expressions are evaluated to 8-bit values and sequentially placed in the hex output file. String constants are placed in the output file according to the rules defined in Section 8.4.2. A DB directive is the only ASM-86 statement that accepts a string constant longer than two bytes. There is no translation from lower to upper case within strings. Multiple expressions or constants, separated by commas, may be added to the definition, but may not exceed the physical line length.

Use an optional symbol to reference the defined data area throughout the program. The symbol has four attributes: the Segment and Offset attributes determine the symbol's memory reference, the Type attribute specifies single bytes, and Length tells the number of bytes (allocation units) reserved.

The following statements show DB directives with symbols:

```
005F 43502F4D2073 TEXT     DB      'CP/M system',0
     797374656D00
006B E1           AA       DB      'a' + 80H
006C 0102030405   X        DB      1,2,3,4,5
                                   .
                                   .
                                   .
0071 B90C00                MOV     CX,LENGTH TEXT
```

9.9 The DW Directive

```
[symbol] DW <numeric expression>[,<numeric expression>..]
[symbol] DW <string constant>[,<string constant>...]
```

The DW directive initializes two-byte words of storage. String constants longer than two characters are illegal. Otherwise, DW uses the same procedure to initialize storage as DB. The following are examples of DW statements:

```
0074 0000         CNTR     DW      0
0076 63C166C169C1 JMPTAB   DW      SUBR1,SUBR2,SUBR3
007C 010002000300          DW      1,2,3,4,5,6
     040005000600
```

9.10 The DD Directive

```
[symbol] DD <numeric expression>[,<numeric expression>..]
```

The DD directive initializes four bytes of storage. The Offset attribute of the address expression is stored in the two lower bytes, the Segment attribute in the two upper bytes. Otherwise, DD follows the same procedure as DB. For example:

```
  1234                         CSEG    1234H
                                       .
                                       .
                                       .
0000 6CC134126FC1 LONG_JMPTAB  DD      ROUT1,ROUT2
     3412
0008 72C1341275C1              DD      ROUT3,ROUT4
     3412
```

9.11 The RS Directive

[symbol] RS <numeric expression>

The RS directive allocates storage in memory but does not initialize it. The numeric expression gives the number of bytes to be reserved. An RS statement does not give a byte attribute to the optional symbol. For example:

```
0010        BUF   RS   80
0060              RS   4000H
4060              RS   1
```

9.12 The RB Directive

[symbol] RB <numeric expression>

The RB directive allocates byte storage in memory without any initialization. This directive is identical to the RS directive except that it does give the byte attribute.

9.13 The RW Directive

[symbol] RW <numeric expression>

The RW directive allocates two-byte word storage in memory but does not initialize it. The numeric expression gives the number of words to be reserved. For example:

```
4061        BUFF  RW   128
4161              RW   4000H
C161              RW   1
```

9.14 The TITLE Directive

TITLE <string constant>

ASM-86 prints the string constant defined by a TITLE directive statement at the top of each printout page in the listing file. The title character string should not exceed 30 characters. For example:

```
TITLE 'CP/M monitor'
```

9.15 The PAGESIZE Directive

PAGESIZE <numeric expression>

The PAGESIZE directive defines the number of lines to be included on each printout page. The default pagesize is 66.

9.16 The PAGEWIDTH Directive

PAGEWIDTH <numeric expression>

The PAGEWIDTH directive defines the number of columns printed across the page when the listing file is output. The default pagewidth is 120 unless the listing is routed directly to the terminal; then the default pagewidth is 79.

9.17 The EJECT Directive

EJECT

The EJECT directive performs a page eject during printout. The EJECT directive itself is printed on the first line of the next page.

9.18 The SIMFORM Directive

SIMFORM

The SIMFORM directive replaces a form-feed (FF) character in the print file with the correct number of line-feeds (LF). Use this directive when printing out on a printer unable to interpret the form-feed character.

9.19 The NOLIST and LIST Directives

NOLIST
LIST

The NOLIST directive blocks the printout of the following lines. Restart the listing with a LIST directive.

SECTION 10

THE ASM-86 INSTRUCTION SET

10.1 Introduction

The ASM-86 instruction set includes all 8086 machine instructions. The general syntax for instruction statements is given in Section 8.7. The following sections define the specific syntax and required operand types for each instruction, without reference to labels or comments. The instruction definitions are presented in tables for easy reference. For a more detailed description of each instruction, see Intel's MCS-86 Assembly Language Reference Manual. For descriptions of the instruction bit patterns and operations, see Intel's MCS-86 User's Manual.

The instruction-definition tables present ASM-86 instruction statements as combinations of mnemonics and operands. A mnemonic is a symbolic representation for an instruction, and its operands are its required parameters. Instructions can take zero, one or two operands. When two operands are specified, the left operand is the instruction's destination operand, and the two operands are separated by a comma.

The instruction-definition tables organize ASM-86 instructions into functional groups. Within each table, the instructions are listed alphabetically. Table 10-1 shows the symbols used in the instruction-definition tables to define operand types.

Table 10-1. Operand Type Symbols

Symbol	Operand Type
numb	any NUMERIC expression
numb8	any NUMERIC expression which evaluates to an 8-bit number
acc	accumulator register, AX or AL
reg	any general purpose register, not segment register
reg16	a 16-bit general purpose register, not segment register
segreg	any segment register: CS, DS, SS, or ES

Table 10-1. (continued)

Symbol	Operand Type
mem	any ADDRESS expression, with or without base- and/or index-addressing modes, such as: variable variable+3 variable[bx] variable[SI] variable[BX+SI] [BX] [BP+DI]
simpmem	any ADDRESS expression WITHOUT base- and index- addressing modes, such as: variable variable+4
mem|reg	any expression symbolized by "reg" or "mem"
mem|reg16	any expression symbolized by "mem|reg", but must be 16 bits
label	any ADDRESS expression which evaluates to a label
lab8	any "label" which is within +/- 128 bytes distance from the instruction

The 8086 CPU has nine single-bit Flag registers which reflect the state of the CPU. The user cannot access these registers directly, but can test them to determine the effects of an executed instruction upon an operand or register. The effects of instructions on Flag registers are also described in the instruction-definition tables, using the symbols shown in Table 10-2 to represent the nine Flag registers.

Table 10-2. Flag Register Symbols

AF	Auxiliary-Carry-Flag
CF	Carry-Flag
DF	Direction-Flag
IF	Interrupt-Enable-Flag
OF	Overflow-Flag
PF	Parity-Flag
SF	Sign-Flag
TF	Trap-Flag
ZF	Zero-Flag

10.2 Data Transfer Instructions

There are four classes of data transfer operations: general purpose, accumulator specific, address-object and flag. Only SAHF and POPF affect flag settings. Note in Table 10-3 that if acc = AL, a byte is transferred, but if acc = AX, a word is transferred.

Table 10-3. Data Transfer Instructions

Syntax		Result
IN	acc,numb8|numb16	transfer data from input port given by numb8 or numb16 (0-255) to accumulator
IN	acc,DX	transfer data from input port given by DX register (0-0FFFFH) to accumulator
LAHF		transfer flags to the AH register
LDS	reg16,mem	transfer the segment part of the memory address (DWORD variable) to the DS segment register, transfer the offset part to a general purpose 16-bit register
LEA	reg16,mem	transfer the offset of the memory address to a (16-bit) register
LES	reg16,mem	transfer the segment part of the memory address to the ES segment register, transfer the offset part to a 16-bit general purpose register
MOV	reg,mem|reg	move memory or register to register
MOV	mem|reg,reg	move register to memory or register

Table 10-3. (continued)

Syntax		Result
MOV	mem|reg,numb	move immediate data to memory or register
MOV	segreg,mem|reg16	move memory or register to segment register
MOV	mem|reg16,segreg	move segment register to memory or register
OUT	numb8|numb16,acc	transfer data from accumulator to output port (0-255) given by numb8 or numb16
OUT	DX,acc	transfer data from accumulator to output port (0-0FFFFH) given by DX register
POP	mem|reg16	move top stack element to memory or register
POP	segreg	move top stack element to segment register; note that CS segment register not allowed
POPF		transfer top stack element to flags
PUSH	mem|reg16	move memory or register to top stack element
PUSH	segreg	move segment register to top stack element
PUSHF		transfer flags to top stack element
SAHF		transfer the AH register to flags
XCHG	reg,mem|reg	exchange register and memory or register
XCHG	mem|reg,reg	exchange memory or register and register
XLAT	mem|reg	perform table lookup translation, table given by "mem|reg", which is always BX. Replaces AL with AL offset from BX.

10.3 Arithmetic, Logical, and Shift Instructions

The 8086 CPU performs the four basic mathematical operations in several different ways. It supports both 8- and 16-bit operations and also signed and unsigned arithmetic.

Six of the nine flag bits are set or cleared by most arithmetic operations to reflect the result of the operation. Table 10-4 summarizes the effects of arithmetic instructions on flag bits. Table 10-5 defines arithmetic instructions and Table 10-6 logical and shift instructions.

Table 10-4. Effects of Arithmetic Instructions on Flags

CF is set if the operation resulted in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; otherwise CF is cleared.

AF is set if the operation resulted in a carry out of (from addition) or a borrow into (from subtraction) the low-order four bits of the result; otherwise AF is cleared.

ZF is set if the result of the operation is zero; otherwise ZF is cleared.

SF is set if the result is negative.

PF is set if the modulo 2 sum of the low-order eight bits of the result of the operation is 0 (even parity); otherwise PF is cleared (odd parity).

OF is set if the operation resulted in an overflow; the size of the result exceeded the capacity of its destination.

Table 10-5. Arithmetic Instructions

	Syntax	Result
AAA		adjust unpacked BCD (ASCII) for addition - adjusts AL
AAD		adjust unpacked BCD (ASCII) for division - adjusts AL
AAM		adjust unpacked BCD (ASCII) for multiplication - adjusts AX
AAS		adjust unpacked BCD (ASCII) for subtraction - adjusts AL
ADC	reg,mem|reg	add (with carry) memory or register to register
ADC	mem|reg,reg	add (with carry) register to memory or register
ADC	mem|reg,numb	add (with carry) immediate data to memory or register
ADD	reg,mem|reg	add memory or register to register
ADD	mem|reg,reg	add register to memory or register
ADD	mem|reg,numb	add immediate data to memory or register
CBW		convert byte in AL to word in AH by sign extension
CWD		convert word in AX to double word in DX/AX by sign extension
CMP	reg,mem|reg	compare register with memory or register
CMP	mem|reg,reg	compare memory or register with register
CMP	mem|reg,numb	compare data constant with memory or register
DAA		decimal adjust for addition, adjusts AL
DAS		decimal adjust for subtraction, adjusts AL
DEC	mem|reg	subtract 1 from memory or register

Table 10-5. (continued)

<table>
<tr><th></th><th>Syntax</th><th>Result</th></tr>
<tr><td>INC</td><td>mem|reg</td><td>add 1 to memory or register</td></tr>
<tr><td>DIV</td><td>mem|reg</td><td>divide (unsigned) accumulator (AX or AL) by memory or register. If byte results, AL = quotient, AH = remainder. If word results, AX = quotient, DX = remainder</td></tr>
<tr><td>IDIV</td><td>mem|reg</td><td>divide (signed) accumulator (AX or AL) by memory or register - quotient and remainder stored as in DIV</td></tr>
<tr><td>IMUL</td><td>mem|reg</td><td>multiply (signed) memory or register by accumulator (AX or AL) - if byte, results in AH, AL. If word, results in DX, AX</td></tr>
<tr><td>MUL</td><td>mem|reg</td><td>multiply (unsigned) memory or register by accumulator (AX or AL) - results stored as in IMUL</td></tr>
<tr><td>NEG</td><td>mem|reg</td><td>two's complement memory or register</td></tr>
<tr><td>SBB</td><td>reg,mem|reg</td><td>subtract (with borrow) memory or register from register</td></tr>
<tr><td>SBB</td><td>mem|reg,reg</td><td>subtract (with borrow) register from memory or register</td></tr>
<tr><td>SBB</td><td>mem|reg,numb</td><td>subtract (with borrow) immediate data from memory or register</td></tr>
<tr><td>SUB</td><td>reg,mem|reg</td><td>subtract memory or register from register</td></tr>
<tr><td>SUB</td><td>mem|reg,reg</td><td>subtract register from memory or register</td></tr>
<tr><td>SUB</td><td>mem|reg,numb</td><td>subtract data constant from memory or register</td></tr>
</table>

Table 10-6. Logic and Shift Instructions

	Syntax	Result
AND	reg,mem|reg	perform bitwise logical "and" of a register and memory register
AND	mem|reg,reg	perform bitwise logical "and" of memory register and register
AND	mem|reg,numb	perform bitwise logical "and" of memory register and data constant
NOT	mem|reg	form ones complement of memory or register
OR	reg,mem|reg	perform bitwise logical "or" of a register and memory register
OR	mem|reg,reg	perform bitwise logical "or" of memory register and register
OR	mem|reg,numb	perform bitwise logical "or" of memory register and data constant
RCL	mem|reg,1	rotate memory or register 1 bit left through carry flag
RCL	mem|reg,CL	rotate memory or register left through carry flag, number of bits given by CL register
RCR	mem|reg,1	rotate memory or register 1 bit right through carry flag
RCR	mem|reg,CL	rotate memory or register right through carry flag, number of bits given by CL register
ROL	mem|reg,1	rotate memory or register 1 bit left
ROL	mem|reg,CL	rotate memory or register left, number of bits given by CL register
ROR	mem|reg,1	rotate memory or register 1 bit right
ROR	mem|reg,CL	rotate memory or register right, number of bits given by CL register
SAL	mem|reg,1	shift memory or register 1 bit left, shift in low-order zero bits

Table 10-6. (continued)

	Syntax	Result
SAL	mem|reg,CL	shift memory or register left, number of bits given by CL register, shift in low-order zero bits
SAR	mem|reg,1	shift memory or register 1 bit right, shift in high-order bits equal to the original high-order bit
SAR	mem|reg,CL	shift memory or register right, number of bits given by CL register, shift in high-order bits equal to the original high-order bit
SHL	mem|reg,1	shift memory or register 1 bit left, shift in low-order zero bits - note that SHL is a different mnemonic for SAL
SHL	mem|reg,CL	shift memory or register left, number of bits given by CL register, shift in low-order zero bits - note that SHL is a different mnemonic for SAL
SHR	mem|reg,1	shift memory or register 1 bit right, shift in high-order zero bits
SHR	mem|reg,CL	shift memory or register right, number of bits given by CL register, shift in high-order zero bits
TEST	reg,mem|reg	perform bitwise logical "and" of a register and memory or register - set condition flags but do not change destination
TEST	mem|reg,reg	perform bitwise logical "and" of memory register and register - set condition flags but do not change destination
TEST	mem|reg,numb	perform bitwise logical "and" - test of memory register and data constant - set condition flags but do not change destination

Table 10-6. (continued)

	Syntax	Result
XOR	reg,mem|reg	perform bitwise logical "exclusive OR" of a register and memory or register
XOR	mem|reg,reg	perform bitwise logical "exclusive OR" of memory register and register
XOR	mem|reg,numb	perform bitwise logical "exclusive OR" of memory register and data constant

10.4 String Instructions

String instructions take one or two operands. The operands specify only the operand type, determining whether operation is on bytes or words. If there are two operands, the source operand is addressed by the SI register and the destination operand is addressed by the DI register. The DI and SI registers are always used for addressing. Note that for string operations, destination operands addressed by DI must always reside in the Extra Segment (ES).

Table 10-7. String Instructions

	Syntax	Result
CMPS	mem|reg,mem|reg	subtract source from destination, affect flags, but do not return result.
LODS	mem|reg	transfer a byte or word from the source operand to the accumulator.
MOVS	mem|reg,mem|reg	move 1 byte (or word) from source to destination.
SCAS	mem|reg	subtract destination operand from accumulator (AX or AL), affect flags, but do not return result.
STOS	mem|reg	transfer a byte or word from accumulator to the destination operand.

Table 10-8 defines prefixes for string instructions. A prefix repeats its string instruction the number of times contained in the CX register, which is decremented by 1 for each iteration. Prefix mnemonics precede the string instruction mnemonic in the statement line as shown in Section 8.8.

Table 10-8. Prefix Instructions

Syntax	Result
REP	repeat until CX register is zero
REPZ	repeat until CX register is zero and zero flag (ZF) is not zero
REPE	equal to "REPZ"
REPNZ	repeat until CX register is zero and zero flag (ZF) is zero
REPNE	equal to "REPNZ"

10.5 Control Transfer Instructions

There are four classes of control transfer instructions:

- calls, jumps, and returns
- conditional jumps
- iterational control
- interrupts

All control transfer instructions cause program execution to continue at some new location in memory, possibly in a new code segment. The transfer may be absolute or depend upon a certain condition. Table 10-9 defines control transfer instructions. In the definitions of conditional jumps, "above" and "below" refer to the relationship between unsigned values, and "greater than" and "less than" refer to the relationship between signed values.

Table 10-9. Control Transfer Instructions

	Syntax	Result
CALL	label	push the offset address of the next instruction on the stack, jump to the target label
CALL	mem|reg16	push the offset address of the next instruction on the stack, jump to location indicated by contents of specified memory or register
CALLF	label	push CS segment register on the stack, push the offset address of the next instruction on the stack (after CS), jump to the target label
CALLF	mem	push CS register on the stack, push the offset address of the next instruction on the stack, jump to location indicated by contents of specified double word in memory
INT	numb8	push the flag registers (as in PUSHF), clear TF and IF flags, transfer control with an indirect call through any one of the 256 interrupt-vector elements - uses three levels of stack
INTO		if OF (the overflow flag) is set, push the flag registers (as in PUSHF), clear TF and IF flags, transfer control with an indirect call through interrupt-vector element 4 (location 10H) - if the OF flag is cleared, no operation takes place
IRET		transfer control to the return address saved by a previous interrupt operation, restore saved flag registers, as well as CS and IP - pops three levels of stack
JA	lab8	jump if "not below or equal" or "above" ((CF or ZF)=0)

Table 10-9. (continued)

Syntax		Result
JAE	lab8	jump if "not below" or "above or equal" (CF=0)
JB	lab8	jump if "below" or "not above or equal" (CF=1)
JBE	lab8	jump if "below or equal" or "not above" ((CF or ZF)=1)
JC	lab8	same as "JB"
JCXZ	lab8	jump to target label if CX register is zero
JE	lab8	jump if "equal" or "zero" (ZF=1)
JG	lab8	jump if "not less or equal" or "greater" (((SF xor OF) or ZF)=0)
JGE	lab8	jump if "not less" or "greater or equal" ((SF xor OF)=0)
JL	lab8	jump if "less" or "not greater or equal" ((SF xor OF)=1)
JLE	lab8	jump if "less or equal" or "not greater" (((SF xor OF) or ZF)=1)
JMP	label	jump to the target label
JMP	mem|reg16	jump to location indicated by contents of specified memory or register
JMPF	label	jump to the target label possibly in another code segment
JMPS	lab8	jump to the target label within +/- 128 bytes from instruction
JNA	lab8	same as "JBE"
JNAE	lab8	same as "JB"
JNB	lab8	same as "JAE"
JNBE	lab8	same as "JA"
JNC	lab8	same as "JNB"

Table 10-9. (continued)

Syntax		Result
JNE	lab8	jump if "not equal" or "not zero" (ZF=0)
JNG	lab8	same as "JLE"
JNGE	lab8	same as "JL"
JNL	lab8	same as "JGE"
JNLE	lab8	same as "JG"
JNO	lab8	jump if "not overflow" (OF=0)
JNP	lab8	jump if "not parity" or "parity odd"
JNS	lab8	jump if "not sign"
JNZ	lab8	same as "JNE"
JO	lab8	jump if "overflow" (OF=1)
JP	lab8	jump if "parity" or "parity even" (PF=1)
JPE	lab8	same as "JP"
JPO	lab8	same as "JNP"
JS	lab8	jump if "sign" (SF=1)
JZ	lab8	same as "JE"
LOOP	lab8	decrement CX register by one, jump to target label if CX is not zero
LOOPE	lab8	decrement CX register by one, jump to target label if CX is not zero and the ZF flag is set - "loop while zero" or "loop while equal"
LOOPNE	lab8	decrement CX register by one, jump to target label if CX is not zero and ZF flag is cleared - "loop while not zero" or "loop while not equal"
LOOPNZ	lab8	same as "LOOPNE"
LOOPZ	lab8	same as "LOOPE"

Table 10-9. (continued)

Syntax		Result
RET		return to the return address pushed by a previous CALL instruction, increment stack pointer by 2
RET	numb	return to the address pushed by a previous CALL, increment stack pointer by 2+numb
RETF		return to the address pushed by a previous CALLF instruction, increment stack pointer by 4
RETF	numb	return to the address pushed by a previous CALLF instruction, increment stack pointer by 4+numb

10.6 Processor Control Instructions

Processor control instructions manipulate the flag registers. Moreover, some of these instructions can synchronize the 8086 CPU with external hardware.

Table 10-10. Processor Control Instructions

Syntax		Results
CLC		clear CF flag
CLD		clear DF flag, causing string instructions to auto-increment the operand pointers
CLI		clear IF flag, disabling maskable external interrupts
CMC		complement CF flag
ESC	numb8,mem|reg	do no operation other than compute the effective address and place it on the address bus (ESC is used by the 8087 numeric co-processor), "numb8" must be in the range 0, 63

Table 10-10. (continued)

Syntax	Results
LOCK	PREFIX instruction, cause the 8086 processor to assert the "bus-lock" signal for the duration of the operation caused by the following instruction - the LOCK prefix instruction may precede any other instruction - buslock prevents co-processors from gaining the bus; this is useful for shared-resource semaphores
HLT	cause 8086 processor to enter halt state until an interrupt is recognized
STC	set CF flag
STD	set DF flag, causing string instructions to auto-decrement the operand pointers
STI	set IF flag, enabling maskable external interrupts
WAIT	cause the 8086 processor to enter a "wait" state if the signal on its "TEST" pin is not asserted

SECTION 11

CODE-MACRO FACILITIES

11.1 Introduction to Code-macros

ASM-86 does not support traditional assembly-language macros, but it does allow the user to define his own instructions by using the Code-macro directive. Like traditional macros, code-macros are assembled wherever they appear in assembly language code, but there the similarity ends. Traditional macros contain assembly language instructions, but a code-macro contains only code-macro directives. Macros are usually defined in the user's symbol table; ASM-86 code-macros are defined in the assembler's symbol table. A macro simplifies using the same block of instructions over and over again throughout a program, but a code-macro sends a bit stream to the output file and in effect adds a new instruction to the assembler.

Because ASM-86 treats a code-macro as an instruction, you can invoke code-macros by using them as instructions in your program. The example below shows how MAC, an instruction defined by a code-macro, can be invoked.

```
          .
          .
          .
     XCHG BX,WORD3
     MAC  PAR1,PAR2
     MUL  AX,WORD4
          .
          .
          .
```

Note that MAC accepts two operands. When MAC was defined, these two operands were also classified as to type, size, and so on by defining MAC's formal parameters. The names of formal parameters are not fixed. They are stand-ins which are replaced by the names or values supplied as operands when the code-macro is invoked. Thus formal parameters "hold the place" and indicate where and how the operands are to be used.

The definition of a code-macro starts with a line specifying its name and its formal parameters, if any:

```
CodeMacro <name> [<formal parameter list>]
```

where the optional <formal parameter list> is defined:

```
<formal name>:<specifier letter>[<modifier letter>][<range>]
```

As stated above, the formal name is not fixed, but a place holder. If formal parameter list is present, the specifier letter is required and the modifier letter is optional. Possible specifiers are A, C, D, E, M, R, S, and X. Possible modifier letters are b, d, w, and sb. The assembler ignores case except within strings, but for clarity, this section shows specifiers in upper-case and modifiers in lower-case. Following sections describe specifiers, modifiers, and the optional range in detail.

The body of the code-macro describes the bit pattern and formal parameters. Only the following directives are legal within code-macros:

```
SEGFIX
NOSEGFIX
MODRM
RELB
RELW
DB
DW
DD
DBIT
```

These directives are unique to code-macros, and those which appear to duplicate ASM-86 directives (DB, DW, and DD) have different meanings in code-macro context. These directives are discussed in detail in later sections. The definition of a code-macro ends with a line:

```
EndM
```

CodeMacro, EndM, and the code-macro directives are all reserved words. Code-macro definition syntax is defined in Backus-Naur-like form in Appendix G. The following examples are typical code-macro definitions.

```
CodeMacro AAA
  DB 37H
EndM

CodeMacro DIV divisor:Eb
  SEGFIX divisor
  DB     6FH
  MODRM  divisor
EndM

CodeMacro ESC opcode:Db(0,63),src:Eb
  SEGFIX src
  DBIT 5(1BH),3(opcode(3))
  MODRM  opcode,src
EndM
```

11.2 Specifiers

Every formal parameter must have a specifier letter that indicates what type of operand is needed to match the formal parameter. Table 11-1 defines the eight possible specifier letters.

Table 11-1. Code-macro Operand Specifiers

Letter	Operand Type
A	Accumulator register, AX or AL.
C	Code, a label expression only.
D	Data, a number to be used as an immediate value.
E	Effective address, either an M (memory address) or an R (register).
M	Memory address. This can be either a variable or a bracketed register expression.
R	A general register only.
S	Segment register only.
X	A direct memory reference.

11.3 Modifiers

The optional modifier letter is a further requirement on the operand. The meaning of the modifier letter depends on the type of the operand. For variables, the modifier requires the operand to be of type: "b" for byte, "w" for word, "d" for double-word and "sb" for signed byte. For numbers, the modifiers require the number to be of a certain size: "b" for -256 to 255 and "w" for other numbers. Table 11-2 summarizes code-macro modifiers.

Table 11-2. Code-macro Operand Modifiers

Variables		Numbers	
Modifier	Type	Modifier	Size
b	byte	b	-256 to 255
w	word	w	anything else
d	dword		
sb	signed byte		

11.4 Range Specifiers

The optional range is specified within parentheses by either one expression or two expressions separated by a comma. The following are valid formats:

```
(numberb)
(register)
(numberb,numberb)
(numberb,register)
(register,numberb)
(register,register)
```

Numberb is 8-bit number, not an address. The following example specifies that the input port must be identified by the DX register:

```
CodeMacro IN dst:Aw,port:Rw(DX)
```

The next example specifies that the CL register is to contain the "count" of rotation:

```
CodeMacro ROR dst:Ew,count:Rb(CL)
```

The last example specifies that the "opcode" is to be immediate data, and may range from 0 to 63 inclusive:

```
CodeMacro ESC opcode:Db(0,63),adds:Eb
```

11.5 Code-macro Directives

Code-macro directives define the bit pattern and make further requirements on how the operand is to be treated. Directives are reserved words, and those that appear to duplicate assembly language instructions have different meanings within a code-macro definition. Only the nine directives defined here are legal within code-macro definitions.

11.5.1 SEGFIX

If SEGFIX is present, it instructs the assembler to determine whether a segment-override prefix byte is needed to access a given memory location. If so, it is output as the first byte of the instruction. If not, no action is taken. SEGFIX takes the form:

```
SEGFIX <formal name>
```

where <formal name> is the name of a formal parameter which represents the memory address. Because it represents a memory address, the formal parameter must have one of the specifiers E, M or X.

11.5.2 NOSEGFIX

Use NOSEGFIX for operands in instructions that must use the ES register for that operand. This applies only to the destination operand of these instructions: CMPS, MOVS, SCAS, STOS. The form of NOSEGFIX is:

```
NOSEGFIX   segreg,<formname>
```

where segreg is one of the segment registers ES, CS, SS, or DS and <formname> is the name of the memory-address formal parameter, which must have a specifier E, M, or X. No code is generated from this directive, but an error check is performed. The following is an example of NOSEGFIX use:

```
CodeMacro MOVS si_ptr:Ew,di_ptr:Ew
  NOSEGFIX    ES,di_ptr
  SEGFIX      si_ptr
  DB          0A5H
EndM
```

11.5.3 MODRM

This directive intructs the assembler to generate the ModRM byte, which follows the opcode byte in many of the 8086's instructions. The ModRM byte contains either the indexing type or the register number to be used in the instruction. It also specifies which register is to be used, or gives more information to specify an instruction.

The ModRM byte carries the information in three fields: The mod field occupies the two most significant bits of the byte, and combines with the register memory field to form 32 possible values: 8 registers and 24 indexing modes.

The reg field occupies the three next bits following the mod field. It specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the opcode byte.

The register memory field occupies the last three bits of the byte. It specifies a register as the location of an operand, or forms a part of the address-mode in combination with the mod field described above.

For further information of the 8086's instructions and their bit patterns, see Intel's 8086 Assembly Language Programing Manual and the Intel 8086 Family User's Manual. The forms of MODRM are:

```
MODRM  <form name>,<form name>
MODRM  NUMBER7,<form name>
```

where NUMBER7 is a value 0 to 7 inclusive and <form name> is the name of a formal parameter. The following examples show MODRM use:

```
CodeMacro RCR dst:Ew,count:Rb(CL)
  SEGFIX      dst
  DB          0D3H
  MODRM       3,dst
EndM

CodeMacro OR dst:Rw,src:Ew
  SEGFIX      src
  DB          0BH
  MODRM       dst,src
EndM
```

11.5.4 RELB and RELW

These directives, used in IP-relative branch instructions, instruct the assembler to generate displacement between the end of the instruction and the label which is supplied as an operand. RELB generates one byte and RELW two bytes of displacement. The directives the following forms:

```
RELB <form name>
RELW <form name>
```

where <form name> is the name of a formal parameter with a "C" (code) specifier. For example:

```
CodeMacro LOOP place:Cb
  DB         0E2H
  RELB       place
EndM
```

11.5.5 DB, DW and DD

These directives differ from those which occur outside of code-macros. The form of the directives are:

```
DB <form name> | NUMBERB
DW <form name> | NUMBERW
DD <form name>
```

where NUMBERB is a single-byte number, NUMBERW is a two-byte number, and <form name> is a name of a formal parameter. For example:

```
CodeMacro XOR dst:Ew,src:Db
  SEGFIX       dst
  DB           81H
  MODRM        6,dst
  DW           src
EndM
```

11.5.6 DBIT

This directive manipulates bits in combinations of a byte or less. The form is:

```
DBIT <field description>[,<field description>]
```

where a <field description>, has two forms:

```
<number><combination>
<number>(<form name>(<rshift>))
```

where <number> ranges from 1 to 16, and specifies the number of bits to be set. <combination> specifies the desired bit combination. The total of all the <number>s listed in the field descriptions must not exceed 16. The second form shown above contains <form name>, a formal parameter name that instructs the assembler to put a certain number in the specified position. This number normally refers to the register specified in the first line of the code-macro. The numbers used in this special case for each register are:

```
AL:   0
CL:   1
DL:   2
BL:   3
AH:   4
CH:   5
DH:   6
BH:   7
AX:   0
CX:   1
DX:   2
BX:   3
SP:   4
BP:   5
SI:   6
DI:   7
ES:   0
CS:   1
SS:   2
DS:   3
```

<rshift>, which is contained in the innermost parentheses, specifies a number of right shifts. For example, "0" specifies no shift, "1" shifts right one bit, "2" shifts right two bits, and so on. The definition below uses this form.

```
CodeMacro DEC dst:Rw
  DBIT 5(9H),3(dst(0))
EndM
```

The first five bits of the byte have the value 9H. If the remaining bits are zero, the hex value of the byte will be 48H. If the instruction:

```
DEC     DX
```

is assembled and DX has a value of 2H, then 48H + 2H = 4AH, which is the final value of the byte for execution. If this sequence had been present in the definition:

```
DBIT 5(9H),3(dst(1))
```

then the register number would have been shifted right once and the result would had been 48H + 1H = 49H, which is erroneous.

SECTION 12

DDT-86

12.1 DDT-86 Operation

The DDT-86™ program allows the user to test and debug programs interactively in a MP/M-86 environment. The reader should be familiar with the 8086 processor, ASM-86 and the MP/M-86 operating system as described in the MP/M-86 System Guide.

12.1.1 Invoking DDT-86

Invoke DDT-86 by entering one of the following commands:

```
DDT86
DDT86 filename
```

The first command simply loads and executes DDT-86. After displaying its sign-on message and prompt character, - , DDT-86 is ready to accept operator commands. The second command is similar to the first, except that after DDT-86 is loaded it loads the file specified by filename. If the file type is omitted from filename, .CMD is assumed. Note that DDT-86 cannot load a file of type .H86. The second form of the invoking command is equivalent to the sequence:

A>**DDT86**
DDT86 x.x
-Efilename

At this point, the program that was loaded is ready for execution.

12.1.2 DDT-86 Command Conventions

When DDT-86 is ready to accept a command, it prompts the operator with a hyphen, -. In response, the operator can type a command line or a CONTROL-C or ↑C to end the debugging session (see Section 12.1.4). A command line may have up to 64 characters, and must be terminated with a carriage return. While entering the command, use standard CP/M line-editing functions (↑X, ↑H, ↑R, etc.) to correct typing errors. DDT-86 does not process the command line until a carriage return is entered.

The first character of each command line determines the command action. Table 12-1 summarizes DDT-86 commands. DDT-86 commands are defined individually in Section 12.2.

Table 12-1. DDT-86 Command Summary

Command	Action
A	enter assembly language statements
D	display memory in hexadecimal and ASCII
E	load program for execution
F	fill memory block with a constant
G	begin execution with optional breakpoints
H	hexadecimal arithmetic
I	set up file control block and command tail
L	list memory using 8086 mnemonics
M	move memory block
R	read disk file into memory
S	set memory to new values
T	trace program execution
U	untraced program monitoring
V	show memory layout of disk file read
W	write contents of memory block to disk
X	examine and modify CPU state

The command character may be followed by one or more arguments, which may be hexadecimal values, file names or other information, depending on the command. Arguments are separated from each other by commas or spaces. No spaces are allowed between the command character and the first argument.

12.1.3 Specifying a 20-Bit Address

Most DDT-86 commands require one or more addresses as operands. Because the 8086 can address up to 1 megabyte of memory, addresses must be 20-bit values. Enter a 20-bit address as follows:

```
ssss:oooo
```

where ssss represents an optional 16-bit segment number and oooo is a 16-bit offset. DDT-86 combines these values to produce a 20-bit effective address as follows:

```
 ssss0
+ oooo
 -----
 eeeee
```

The optional value ssss may be a 16-bit hexadecimal value or the name of a segment register. If a segment register name is specified, the value of ssss is the contents of that register in the user's CPU state, as indicated by the X command. If omitted, a default value appropriate to the command being executed, as described in Section 12.4.

12.1.4 Terminating DDT-86

Terminate DDT-86 by typing a ↑C in response to the hyphen prompt. This returns control to the CCP. Note that MP/M-86 does not have the SAVE facility found in CP/M for 8-bit machines. Thus if DDT-86 is used to patch a file, write the file to disk using the W command before exiting DDT-86.

12.1.5 DDT-86 Operation with Interrupts

DDT-86 operates with interrupts enabled or disabled, and preserves the interrupt state of the program being executed under DDT-86. When DDT-86 has control of the CPU, either when it is initially invoked, or when it regains control from the program being tested, the condition of the interrupt flag is the same as it was when DDT-86 was invoked, except for a few critical regions where interrupts are disabled. While the program being tested has control of the CPU, the user's CPU state, which can be displayed with the X command, determines the state of the interrupt flag.

12.2 DDT-86 Commands

This section defines DDT-86 commands and their arguments. DDT-86 commands give the user control of program execution and allow the user to display and modify system memory and the CPU state.

12.2.1 The A (Assemble) Command

The A command assembles 8086 mnemonics directly into memory. The form is:

```
As
```

where s is the 20-bit address where assembly is to start. DDT-86 responds to the A command by displaying the address of the memory location where assembly is to begin. At this point the operator enters assembly language statements as described in Section 4 on Assembly Language Syntax. When a statement is entered, DDT-86 converts it to binary, places the value(s) in memory, and displays the address of the next available memory location. This process continues until the user enters a blank line or a line containing only a period.

DDT-86 responds to invalid statements by displaying a question mark, ? , and redisplaying the current assembly address.

12.2.2 The D (Display) Command

The D command displays the contents of memory as 8-bit or 16-bit hexadecimal values and in ASCII. The forms are:

```
D
Ds
Ds,f
DW
DWs
DWs,f
```

where s is the 20-bit address where the display is to start, and f is the 16-bit offset within the segment specified in s where the display is to finish.

Memory is displayed on one or more display lines. Each display line shows the values of up to 16 memory locations. For the first three forms, the display line appears as follows:

```
ssss:oooo bb bb . . . bb cc . . . c
```

where ssss is the segment being displayed and oooo is the offset within segment ssss. The bb's represent the contents of the memory locations in hexadecimal, and the c's represent the contents of memory in ASCII. Any non-graphic ASCII characters are represented by periods.

In response to the first form shown above, DDT-86 displays memory from the current display address for 12 display lines. The response to the second form is similar to the first, except that the display address is first set to the 20-bit address s. The third form displays the memory block between locations s and f. The next three forms are analogous to the first three, except that the contents of memory are displayed as 16-bit values, rather than 8-bit values, as shown below:

```
ssss:oooo wwww wwww . . . wwww cccc . . . cc
```

During a long display, the D command may be aborted by typing any character at the console.

12.2.3 The E (Load for Execution) Command

The E command loads a file into memory so that a subsequent G, T or U command can begin program execution. The E command takes the form:

```
E<filename>
```

where <filename> is the name of the file to be loaded. If no file type is specified, .CMD is assumed. The contents of the user segment registers and IP register are altered according to the information in the header of the file loaded.

An E command releases any blocks of memory allocated by any previous E or R commands or by programs executed under DDT-86. Thus only one file at a time may be loaded for execution.

When the load is complete, DDT-86 displays the start and end addresses of each segment in the file loaded. Use the V command to redisplay this information at a later time.

If the file does not exist or cannot be successfully loaded in the available memory, DDT-86 issues an error message.

12.2.4 The F (Fill) Command

The F command fills an area of memory with a byte or word constant. The forms are:

```
Fs,f,b
FWs,f,w
```

where s is a 20-bit starting address of the block to be filled, and f is a 16-bit offset of the final byte of the block within the segment specified in s.

In response to the first form, DDT-86 stores the 8-bit value b in locations s through f. In the second form, the 16-bit value w is stored in locations s through f in standard form, low 8 bits first followed by high 8 bits.

If s is greater than f or the value b is greater than 255, DDT-86 responds with a question mark. DDT-86 issues an error message if the value stored in memory cannot be read back successfully, indicating faulty or non-existent RAM at the location indicated.

12.2.5 The G (Go) Command

The G command transfers control to the program being tested, and optionally sets one or two breakpoints. The forms are:

```
G
G,bl
G,bl,b2
Gs
Gs,bl
Gs,bl,b2
```

where s is a 20-bit address where program execution is to start, and bl and b2 are 20-bit addresses of breakpoints. If no segment value is supplied for any of these three addresses, the segment value defaults to the contents of the CS register.

In the first three forms, no starting address is specified, so DDT-86 derives the 20-bit address from the user's CS and IP registers. The first form transfers control to the user's program without setting any breakpoints. The next two forms respectively set one and two breakpoints before passing control to the user's program. The next three forms are analogous to the first three, except that the user's CS and IP registers are first set to s.

Once control has been transferred to the program under test, it executes in real time until a breakpoint is encountered. At this point, DDT-86 regains control, clears all breakpoints, and indicates the address at which execution of the program under test was interrupted as follows:

```
*ssss:oooo
```

where ssss corresponds to the CS and oooo corresponds to the IP where the break occurred. When a breakpoint returns control to DDT-86, the instruction at the breakpoint address has not yet been executed.

12.2.6 The H (Hexadecimal Math) Command

The H command computes the sum and difference of two 16-bit values. The form is:

```
Ha,b
```

where a and b are the values whose sum and difference are to be computed. DDT-86 displays the sum (ssss) and the difference (dddd) [illegible] line as shown below:

```
ssss dddd
```

12.2.7 The I (Input Command Tail) Command

The I command prepares a file control block and command tail buffer in DDT-86's base page, and copies this information into the base page of the last file loaded with the E command. The form is:

```
I<command tail>
```

where <command tail> is a character string which usually contains one or more filenames. The first filename is parsed into the default file control block at 005CH. The optional second filename (if specified) is parsed into the second part of the default file control block beginning at 006CH. The characters in <command tail> are also copied into the default command buffer at 0080H. The length of <command tail> is stored at 0080H, followed by the character string terminated with a binary zero.

If a file has been loaded with the E command, DDT-86 copies the file control block and command buffer from the base page of DDT-86 to the base page of the program loaded. The location of DDT-86's base page can be obtained from the SS register in the user's CPU state when DDT-86 is invoked. The location of the base page of a program loaded with the E command is the value displayed for DS upon completion of the program load.

12.2.8 The L (List) Command

The L command lists the contents of memory in assembly language. The forms are:

```
L
Ls
Ls,f
```

where s is a 20-bit address where the list is to start, and f is a 16-bit offset within the segment specified in s where the list is to finish.

The first form lists twelve lines of disassembled machine code from the current list address. The second form sets the list address to s and then lists twelve lines of code. The last form lists disassembled code from s through f. In all three cases, the list address is set to the next unlisted location in preparation for a subsequent L command. When DDT-86 regains control from a program being tested (see G, T and U commands), the list address is set to the current value of the CS and IP registers.

Long displays may be aborted by typing any key during the list process. Or, enter ↑S to halt the display temporarily.

The syntax of the assembly language statements produced by the L command is described in Section 10.

12.2.9 The M (Move) Command

The M command moves a block of data values from one area of memory to another. The form is:

```
Ms,f,d
```

where s is the 20-bit starting address of the block to be moved, f is the offset of the final byte to be moved within the segment described by s, and d is the 20-bit address of the first byte of the area to receive the data. If the segment is not specified in d, the same value is used that was used for s. Note that if d is between s and f, part of the block being moved will be overwritten before it is moved, because data is transferred starting from location s.

12.2.10 The R (Read) Command

The R command reads a file into a contiguous block of memory. The form is:

```
R<filename>
```

where <filename> is the name and type of the file to be read.

DDT-86 reads the file into memory and displays the start and end addresses of the block of memory occupied by the file. A V command can redisplay this information at a later time. The default display pointer (for subsequent D commands) is set to the start of the block occupied by the file.

The R command does not free any memory previously allocated by another R or E command. Thus a number of files may be read into memory without overlapping. The number of files which may be loaded is limited to seven, which is the number of memory allocations allowed by the BDOS, minus one for DDT-86 itself.

If the file does not exist or there is not enough memory to load the file, DDT-86 issues an error message.

12.2.11 The S (Set) Command

The S command can change the contents of bytes or words of memory. The forms are:

```
Ss
SWs
```

where s is the 20-bit address where the change is to occur.

DDT-86 displays the memory address and its current contents on the following line. In response to the first form, the display is:

```
ssss:oooo bb
```

and in response to the second form

```
ssss:oooo wwww
```

where bb and wwww are the contents of memory in byte and word formats, respectively.

In response to one of the above displays, the operator may choose to alter the memory location or to leave it unchanged. If a valid hexadecimal value is entered, the contents of the byte (or word) in memory is replaced with the value. If no value is entered, the contents of memory are unaffected and the contents of the next address are displayed. In either case, DDT-86 continues to display successive memory addresses and values until either a period or an invalid value is entered.

DDT-86 issues an error message if the value stored in memory cannot be read back successfully, indicating faulty or non-existent RAM at the location indicated.

12.2.12 The T (Trace) Command

The T command traces program execution for 1 to 0FFFFH program steps. The forms are:

```
T
Tn
TS
TSn
```

where n is the number of instructions to execute before returning control to the console.

Before an instruction is executed, DDT-86 displays the current CPU state and the disassembled instruction. In the first two forms, the segment registers are not displayed, which allows the entire CPU state to be displayed on one line. The next two forms are analogous to the first two, except that all the registers are displayed, which forces the disassembled instruction to be displayed on the next line as in the X command.

In all of the forms, control transfers to the program under test at the address indicated by the CS and IP registers. If n is not specified, one instruction is executed. Otherwise DDT-86 executes n instructions, displaying the CPU state before each step. A long trace may be aborted before n steps have been executed by typing any character at the console.

After a T command, the list address used in the L command is set to the address of the next instruction to be executed.

Note that DDT-86 does not trace through a BDOS interrupt instruction, since DDT-86 itself makes BDOS calls and the BDOS is not reentrant. Instead, the entire sequence of instructions from the BDOS interrupt through the return from BDOS is treated as one traced instruction.

12.2.13 The U (Untrace) Command

The U command is identical to the T command except that the CPU state is displayed only before the first instruction is executed, rather than before every step. The forms are:

```
U
Un
US
USn
```

where n is the number of instructions to execute before returning control to the console. The U command may be aborted before n steps have been executed by striking any key at the console.

12.2.14 The V (Value) Command

The V command displays information about the last file loaded with the E or R commands. The form is:

```
V
```

If the last file was loaded with the E command, the V command displays the start and end addresses of each of the segments contained in the file. If the last file was read with the R command, the V command displays the start and end addresses of the block of memory where the file was read. If neither the R nor E commands have been used, DDT-86 responds to the V command with a question mark, ?.

12.2.15 The W (Write) Command

The W command writes the contents of a contiguous block of memory to disk. The forms are:

```
W<filename>
W<filename>,s,f
```

where <filename> is the filename and file type of the disk file to receive the data, and s and f are the 20-bit first and last addresses of the block to be written. If the segment is not specified in f, DDT-86 uses the same value that was used for s.

If the first form is used, DDT-86 assumes the s and f values from the last file read with an R command. If no file was read with an R command, DDT-86 responds with a question mark, ?. This form is useful for writing out files after patches have been installed, assuming the overall length of the file is unchanged.

In the second form where s and f are specified as 20-bit addresses, the low four bits of s are assumed to be 0. Thus the block being written must always start on a paragraph boundary.

If a file by the name specified in the W command already exists, DDT-86 deletes it before writing a new file.

12.2.16 The X (Examine CPU State) Command

The X command allows the operator to examine and alter the CPU state of the program under test. The forms are:

```
X
Xr
Xf
```

where r is the name of one of the 8086 CPU registers and f is the abbreviation of one of the CPU flags. The first form displays the CPU state in the format:

```
          AX   BX   CX  . . .  SS   ES   IP
--------- xxxx xxxx xxxx . . . xxxx xxxx xxxx
<instruction>
```

The nine hyphens at the beginning of the line indicate the state of the nine CPU flags. Each position may be either a hyphen, indicating that the corresponding flag is not set (0), or a 1-character abbreviation of the flag name, indicating that the flag is set (1). The abbreviations of the flag names are shown in Table 12-2. <instruction> is the disassembled instruction at the next location to be executed, which is indicated by the CS and IP registers.

Table 12-2. Flag Name Abbreviations

Character	Name
O	Overflow
D	Direction
I	Interrupt Enable
T	Trap
S	Sign
Z	Zero
A	Auxiliary Carry
P	Parity
C	Carry

The second form allows the operator to alter the registers in the CPU state of the program being tested. The r following the X is the name of one of the 16-bit CPU registers. DDT-86 responds by displaying the name of the register followed by its current value. If a carriage return is typed, the value of the register is not changed. If a valid value is typed, the contents of the register are changed to that value. In either case, the next register is then displayed. This process continues until a period or an invalid value is entered, or the last register is displayed.

The third form allows the operator to alter one of the flags in the CPU state of the program being tested. DDT-86 responds by displaying the name of the flag followed by its current state. If a carriage return is typed, the state of the flag is not changed. If a valid value is typed, the state of the flag is changed to that value. Only one flag may be examined or altered with each Xf command. Set or reset flags by entering a value of 1 or 0.

12.3 Default Segment Values

DDT-86 has an internal mechanism that keeps track of the current segment value, making segment specification an optional part of a DDT-86 command. DDT-86 divides the command set into two types of commands, according to which segment a command defaults if no segment value is specified in the command line.

The first type of command pertains to the code segment: A (Assemble), L (List Mnemonics) and W (Write). These commands use the internal type-1 segment value if no segment value is specified in the command.

When invoked, DDT-86 sets the type-1 segment value to 0, and changes it when one of the following actions is taken:

- When a file is loaded by an E command, DDT-86 sets the type-1 segment value to the value of the CS register.
- When a file is read by an R command, DDT-86 sets the type-1 segment value to the base segment where the file was read.
- When an X command changes the value of the CS register, DDT-86 changes the type-1 segment value to the new value of the CS register.
- When DDT-86 regains control from a user program after a G, T or U command, it sets the type-1 segment value to the value of the CS register.
- When a segment value is specified explicitly in an A or L command, DDT-86 sets the type-1 segment value to the segment value specified.

The second type of command pertains to the data segment: D (Display), F (Fill), M (Move) and S (Set). These commands use the internal type-2 segment value if no segment value is specified in the command.

When invoked, DDT-86 sets the type-2 segment value to 0, and changes it when one of the following actions is taken:

- When a file is loaded by an E command, DDT-86 sets the type-2 segment value to the value of the DS register.
- When a file is read by an R command, DDT-86 sets the type-2 segment value to the base segment where the file was read.
- When an X command changes the value of the DS register, DDT-86 changes the type-2 segment value to the new value of the DS register.
- When DDT-86 regains control from a user program after a G, T or U command, it sets the type-2 segment value to the value of the DS register.
- When a segment value is specified explicitly in an D, F, M or S command, DDT-86 sets the type-2 segment value to the segment value specified.

When evaluating programs that use identical values in the CS and DS registers, all DDT-86 commands default to the same segment value unless explicitly overridden.

Note that the G (Go) command does not fall into either group, since it defaults to the CS register.

Table 12-3 summarizes DDT-86's default segment values.

Table 12-3. DDT-86 Default Segment Values

Command	type-1	type-2
A	x	
D		x
E	c	c
F		x
G	c	c
H		
I		
L	x	
M		x
R	c	c
S		x
T	c	c
U	c	c
V		
W	x	
X	c	c

x - use this segment default if none specified;
change default if specified explicitly
c - change this segment default

12.4 Assembly Language Syntax for A and L Commands

In general, the syntax of the assembly language statements used in the A and L commands is standard 8086 assembly language. Several minor exceptions are listed below.

- DDT-86 assumes that all numeric values entered are hexadecimal.

- Up to three prefixes (LOCK, repeat, segment override) may appear in one statement, but they all must precede the opcode of the statement. Alternately, a prefix may be entered on a line by itself.

- The distinction between byte and word string instructions is made as follows:

byte	word
LODSB	LODSW
STOSB	STOSW
SCASB	SCASW
MOVSB	MOVSW
CMPSB	CMPSW

- The mnemonics for near and far control transfer instructions are as follows:

short	normal	far
JMPS	JMP	JMPF
	CALL	CALLF
	RET	RETF

- If the operand of a CALLF or JMPF instruction is a 20-bit absolute address, it is entered in the form:

  ssss:oooo

  where ssss is the segment and oooo is the offset of the address.

- Operands that could refer to either a byte or word are ambiguous, and must be preceded either by the prefix "BYTE" or "WORD". These prefixes may be abbreviated to "BY" and "WO". For example:

```
INC     BYTE [BP]
NOT     WORD [1234]
```

  Failure to supply a prefix when needed results in an error message.

- Operands which address memory directly are enclosed in square brackets to distinguish them from immediate values. For example:

```
ADD     AX,5     ;add 5 to register AX
ADD     AX,[5]   ;add the contents of location 5 to AX
```

- The forms of register indirect memory operands are:

```
[pointer register]
[index register]
[pointer register + index register]
```

  where the pointer registers are BX and BP, and the index registers are SI and DI. Any of these forms may be preceded by a numeric offset. For example:

```
ADD     BX,[BP+SI]
ADD     BX,3[BP+SI]
ADD     BX,1D47[BP+SI]
```

12.5 DDT-86 Sample Session

In the following sample session, the user interactively debugs a simple sort program. Comments in italic type explain the steps involved.

Source file of program to test.

```
A>type sort.a86
;
;       simple sort program
;
sort:
        mov     si,0                ;initialize index
        mov     bx,offset nlist     ;bx = base of list
        mov     sw,0                ;clear switch flag
comp:
        mov     al,[bx+si]          ;get byte from list
        cmp     al,1[bx+si]         ;compare with next byte
        jna     inci                ;don't switch if in order
        xchg    al,1[bx+si]         ;do first part of switch
        mov     [bx+si],al          ;do second part
        mov     sw,1                ;set switch flag
inci:
        inc     si                  ;increment index
        cmp     si,count            ;end of list?
        jnz     comp                ;no, keep going
        test    sw,1                ;done - any switches?
        jnz     sort                ;yes, sort some more
done:
        jmp     done                ;get here when list ordered
;
        dseg
        org     100h                ;leave space for base page
;
nlist   db      3,8,4,6,31,6,4,1
count   equ     offset $ - offset nlist
sw      db      0
        end
```

Assemble program.

```
A>asm86 sort

CP/M 8086 ASSEMBLER VER 1.1
END OF PASS 1
END OF PASS 2
END OF ASSEMBLY. NUMBER OF ERRORS:   0
```

Type listing file generated by ASM-86.

```
A>type sort.lst
CP/M ASM86 1.1  SOURCE: SORT.A86                                     PAGE    1


                   ;
                   ;       simple sort program
                   ;
                   sort:
0000 BE0000                mov     si,0                ;initialize index
0003 BB0001                mov     bx,offset nlist     ;bx = base of list
0006 C606080100            mov     sw,0                ;clear switch flag
                   comp:
000B 8A00                  mov     al,[bx+si]          ;get byte from list
000D 3A4001                cmp     al,1[bx+si]         ;compare with next byte
0010 760A                  jna     inci                ;don't switch if in order
0012 864001                xchg    al,1[bx+si]         ;do first part of switch
0015 8800                  mov     [bx+si],al          ;do second part
0017 C606080101            mov     sw,1                ;set switch flag
                   inci:
001C 46                    inc     si                  ;increment index
001D 83FE08                cmp     si,count            ;end of list?
0020 75E9                  jnz     comp                ;no, keep going
0022 F606080101            test    sw,1                ;done - any switches?
0027 75D7                  jnz     sort                ;yes, sort some more
                   done:
0029 E9FDFF                jmp     done                ;get here when list ordered
                   ;
                           dseg
                           org     100h                ;leave space for base page
                   ;
```

```
0100 030804061F06 nlist     db      3,8,4,6,31,6,4,1
     0401
  0008                count    equ     offset $ - offset nlist
0108 00               sw       db      0
                               end


END OF ASSEMBLY. NUMBER OF ERRORS:   0
```

Type symbol table file generated by ASM-86.

```
A>type sort.sym
0000 VARIABLES
0100 NLIST        0108 SW

0000 NUMBERS
0008 COUNT

0000 LABELS
000B COMP         0029 DONE         001C INCI          0000 SORT
```

Type hex file generated by ASM-86.

```
A>type sort.h86
:0400000300000000F9
:1B000081BE0000BB0001C6060801008A003A4001760A864001880C606080l6C
:11001B81014683FE0875E9F60608010175D7E9FDFFEE
:090100820308040610F0604010035
:00000001FF
```

Generate CMD file from .H86 file.

```
A>gencmd sort

BYTES READ     0039
RECORDS WRITTEN 04
```

Invoke DDT-86 and load SORT.CMD.

```
A>ddt86 sort
DDT86 1.0
     START       END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F
```

Display initial register values.

```
-x
          AX   BX   CX   DX   SP   BP   SI   DI   CS   DS   SS   ES   IP
--------- 0000 0000 0000 0000 119E 0000 0000 0000 047D 0480 0491 0480 0000
MOV    SI,0000
```

Disassemble the beginning of the code segment.

```
-l
047D:0000 MOV    SI,0000
047D:0003 MOV    BX,0100
047D:0006 MOV    BYTE [0108],00
047D:000B MOV    AL,[BX+SI]
047D:000D CMP    AL,01[BX+SI]
047D:0010 JBE    001C
047D:0012 XCHG   AL,01[BX+SI]
047D:0015 MOV    [BX+SI],AL
047D:0017 MOV    BYTE [0108],01
047D:001C INC    SI
047D:001D CMP    SI,0008
047D:0020 JNZ    000B
```

Display the start of the data segment.

```
-d100,10f
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00 ................
```

Disassemble the rest of the code.

```
-l
047D:0022 TEST   BYTE [0108],01
047D:0027 JNZ    0000
047D:0029 JMP    0029
047D:002C ADD    [BX+SI],AL
047D:002E ADD    [BX+SI],AL
047D:0030 DAS
047D:0031 ADD    [BX+SI],AL
047D:0033 ??=    6C
047D:0034 POP    ES
047D:0035 ADD    [BX],CL
047D:0037 ADD    [BX+SI],AX
047D:0039 ??=    6F
```

Execute program from IP (=0) setting breakpoint at 29H.

```
-g,29
*047D:0029
```

Breakpoint encountered.

Display sorted list.

```
-d100,10f
0480:0100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Doesn't look good; reload file.

```
-esort
     START      END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F
```

Trace 3 instructions.

```
-t3
           AX   BX   CX   DX   SP   BP   SI   DI   IP
-----Z-P- 0000 0100 0000 0000 119E 0000 0008 0000 0000 MOV   SI,0000
-----Z-P- 0000 0100 0000 0000 119E 0000 0000 0000 0003 MOV   BX,0100
-----Z-P- 0000 0100 0000 0000 119E 0000 0000 0000 0006 MOV   BYTE [0108],00
*047D:000B
```

Trace some more.

```
-t3
           AX   BX   CX   DX   SP   BP   SI   DI   IP
-----Z-P- 0000 0100 0000 0000 119E 0000 0000 0000 000B MOV   AL,[BX+SI]
-----Z-P- 0003 0100 0000 0000 119E 0000 0000 0000 000D CMP   AL,01[BX+SI]
----S-A-C 0003 0100 0000 0000 119E 0000 0000 0000 0010 JBE   001C
*047D:001C
```

Display unsorted list.

```
-d100,10f
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00 ................
```

Display next instructions to be executed.

```
-l
047D:001C INC    SI
047D:001D CMP    SI,0008
047D:0020 JNZ    000B
047D:0022 TEST   BYTE [0108],01
047D:0027 JNZ    0000
047D:0029 JMP    0029
047D:002C ADD    [BX+SI],AL
047D:002E ADD    [BX+SI],AL
047D:0030 DAS
047D:0031 ADD    [BX+SI],AL
047D:0033 ??=    6C
047D:0034 POP    ES
```

Trace some more.

```
-t3
           AX   BX   CX   DX   SP   BP   SI   DI   IP
----S-A-C 0003 0100 0000 0000 119E 0000 0000 0000 001C INC   SI
--------C 0003 0100 0000 0000 119E 0000 0001 0000 001D CMP   SI,0008
----S-APC 0003 0100 0000 0000 119E 0000 0001 0000 0020 JNZ   000B
*047D:000B
```

Display instructions from current IP.

```
-l
047D:000B MOV     AL,[BX+SI]
047D:000D CMP     AL,01[BX+SI]
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029

-t3
           AX   BX   CX   DX   SP   BP   SI   DI   IP
----S-APC 0003 0100 0000 0000 119E 0000 0001 0000 000B MOV     AL,[BX+SI]
----S-APC 0008 0100 0000 0000 119E 0000 0001 0000 000D CMP     AL,01[BX+SI]
--------- 0008 0100 0000 0000 119E 0000 0001 0000 0010 JBE     001C
*047D:0012

-l
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029
047D:002C ADD     [BX+SI],AL
047D:002E ADD     [BX+SI],AL
047D:0030 DAS
```

Go until switch has been performed.

```
-g,20
*047D:0020
```

Display list.

```
-d100,10f
0480:0100 03 04 08 06 1F 06 04 01 01 00 00 00 00 00 00 00 ................
```

Looks like 4 and 8 were switched okay. (And toggle is true.)

```
-t
           AX   BX   CX   DX   SP   BP   SI   DI   IP
----S-APC 0004 0100 0000 0000 119E 0000 0002 0000 0020 JNZ     000B
*047D:000B
```

Display next instructions.

```
-l
047D:000B MOV     AL,[BX+SI]
047D:000D CMP     AL,01[BX+SI]
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029
```

Since switch worked, let's reload and check boundary conditions.

```
-esort
     START      END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F
```

```
        Make it quicker by setting list length to 3.  (Could also have used s47d=1e
-ald    to patch.)
047D:001D cmp si,3
047D:0020
        Display unsorted list.
-d100
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00 ................
0480:0110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0480:0120 00 00 00 00 00 00 00 00 00 00 00 00 00 20 20 20 .............
        Set breakpoint when first 3 elements of list should be sorted.
-g,29
*047D:0029

-d100,10f See if list is sorted.
0480:0100 03 04 06 08 1F 06 04 01 00 00 00 00 00 00 00 00 ................

-esort  Interesting, the fourth element seems to have been sorted in.
     START      END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F

        Let's try again with some tracing.
-ald
047D:001D cmp si,3
047D:0020

-t9
          AX   BX   CX   DX   SP   BP   SI   DI   IP
-----Z-P- 0006 0100 0000 0000 119E 0000 0003 0000 0000 MOV   SI,0000
-----Z-P- 0006 0100 0000 0000 119E 0000 0000 0000 0003 MOV   BX,0100
-----Z-P- 0006 0100 0000 0000 119E 0000 0000 0000 0006 MOV   BYTE [0108],00
-----Z-P- 0006 0100 0000 0000 119E 0000 0000 0000 000B MOV   AL,[BX+SI]
-----Z-P- 0003 0100 0000 0000 119E 0000 0000 0000 000D CMP   AL,01[BX+SI]
----S-A-C 0003 0100 0000 0000 119E 0000 0000 0000 0010 JBE   001C
----S-A-C 0003 0100 0000 0000 119E 0000 0000 0000 001C INC   SI
--------C 0003 0100 0000 0000 119E 0000 0001 0000 001D CMP   SI,0003
----S-A-C 0003 0100 0000 0000 119E 0000 0001 0000 0020 JNZ   000B
*047D:000B

-l
047D:000B MOV   AL,[BX+SI]
047D:000D CMP   AL,01[BX+SI]
047D:0010 JBE   001C
047D:0012 XCHG  AL,01[BX+SI]
047D:0015 MOV   [BX+SI],AL
047D:0017 MOV   BYTE [0108],01
047D:001C INC   SI
047D:001D CMP   SI,0003
047D:0020 JNZ   000B
047D:0022 TEST  BYTE [0108],01
047D:0027 JNZ   0000
047D:0029 JMP   0029

-t3
          AX   BX   CX   DX   SP   BP   SI   DI   IP
----S-A-C 0003 0100 0000 0000 119E 0000 0001 0000 000B MOV   AL,[BX+SI]
----S-A-C 0008 0100 0000 0000 119E 0000 0001 0000 000D CMP   AL,01[BX+SI]
--------- 0008 0100 0000 0000 119E 0000 0001 0000 0010 JBE   001C
*047D:0012

-l
047D:0012 XCHG  AL,01[BX+SI]
047D:0015 MOV   [BX+SI],AL
047D:0017 MOV   BYTE [0108],01
047D:001C INC   SI
047D:001D CMP   SI,0003
047D:0020 JNZ   000B
047D:0022 TEST  BYTE [0108],01
```

```
-t3
           AX   BX   CX   DX   SP   BP   SI   DI   IP
--------- 0008 0100 0000 0000 119E 0000 0001 0000 0012 XCHG   AL,01[BX+SI]
--------- 0004 0100 0000 0000 119E 0000 0001 0000 0015 MOV    [BX+SI],AL
--------- 0004 0100 0000 0000 119E 0000 0001 0000 0017 MOV    BYTE [0108],01
*047D:001C

-d100,10f
0480:0100 03 04 08 06 1F 06 04 01 01 00 00 00 00 00 00 00 ................
```

So far, so good.

```
-t3
           AX   BX   CX   DX   SP   BP   SI   DI   IP
--------- 0004 0100 0000 0000 119E 0000 0001 0000 001C INC    SI
--------- 0004 0100 0000 0000 119E 0000 0002 0000 001D CMP    SI,0003
----S-APC 0004 0100 0000 0000 119E 0000 0002 0000 0020 JNZ    000B
*047D:000B

-l
047D:000B MOV    AL,[BX+SI]
047D:000D CMP    AL,01[BX+SI]
047D:0010 JBE    001C
047D:0012 XCHG   AL,01[BX+SI]
047D:0015 MOV    [BX+SI],AL
047D:0017 MOV    BYTE [0108],01
047D:001C INC    SI
047D:001D CMP    SI,0003
047D:0020 JNZ    000B
047D:0022 TEST   BYTE [0108],01
047D:0027 JNZ    0000
047D:0029 JMP    0029

-t3
           AX   BX   CX   DX   SP   BP   SI   DI   IP
----S-APC 0004 0100 0000 0000 119E 0000 0002 0000 000B MOV    AL,[BX+SI]
----S-APC 0008 0100 0000 0000 119E 0000 0002 0000 000D CMP    AL,01[BX+SI]
--------- 0008 0100 0000 0000 119E 0000 0002 0000 0010 JBE    001C
*047D:0012
```

Sure enough, it's comparing the third and fourth elements of the list.

```
-esort
```

Reload program.

```
     START     END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F

-l
047D:0000 MOV    SI,0000
047D:0003 MOV    BX,0100
047D:0006 MOV    BYTE [0108],00
047D:000B MOV    AL,[BX+SI]
047D:000D CMP    AL,01[BX+SI]
047D:0010 JBE    001C
047D:0012 XCHG   AL,01[BX+SI]
047D:0015 MOV    [BX+SI],AL
047D:0017 MOV    BYTE [0108],01
047D:001C INC    SI
047D:001D CMP    SI,0008
047D:0020 JNZ    000B
```

Patch length.

```
-a1d
047D:001D cmp si,7
047D:0020
```

Try it out.

```
-g,29
*047D:0029
```

```
        See if list is sorted.
-d100,10f
0480:0100 01 03 04 04 06 06 08 1F 00 00 00 00 00 00 00 00 ................

        Looks better; let's install patch in disk file.  To do this, we
-rsort.cmd              must read CMD file including header, so we use R
  START      END        command.
2000:0000 2000:01FF

        First 80h bytes contain header, so code starts at 80h.
-l80
2000:0080 MOV    SI,0000
2000:0083 MOV    BX,0100
2000:0086 MOV    BYTE [0108],00
2000:008B MOV    AL,[BX+SI]
2000:008D CMP    AL,01[BX+SI]
2000:0090 JBE    009C
2000:0092 XCHG   AL,01[BX+SI]
2000:0095 MOV    [BX+SI],AL
2000:0097 MOV    BYTE [0108],01
2000:009C INC    SI
2000:009D CMP    SI,0008
2000:00A0 JNZ    008B

        Install patch.
-a9d
2000:009D cmp si,7

        Write file back to disk.  (Length of file assumed to be unchanged
-wsort.cmd           since no length specified.)

        Reload file.
-esort

    START      END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F

        Verify that patch was installed.
-l
047D:0000 MOV    SI,0000
047D:0003 MOV    BX,0100
047D:0006 MOV    BYTE [0108],00
047D:000B MOV    AL,[BX+SI]
047D:000D CMP    AL,01[BX+SI]
047D:0010 JBE    001C
047D:0012 XCHG   AL,01[BX+SI]
047D:0015 MOV    [BX+SI],AL
047D:0017 MOV    BYTE [0108],01
047D:001C INC    SI
047D:001D CMP    SI,0007
047D:0020 JNZ    000B

        Run it.
-g,29
*047D:0029

        Still looks good.  Ship it!
-d100,10f
0480:0100 01 03 04 04 06 06 08 1F 00 00 00 00 00 00 00 00 ................
-^C
A>
```

APPENDIX A

ASM-86 INVOCATION

Command: ASM86

Syntax: ASM86 <filename> { $ <parameters> }

where

<filename> is the 8086 assembly source file (drive and extension are optional)

<parameters> are a one-letter type followed by a one-letter device from the table below.

Default file extension: .A86

Parameters:

form: $ Td where T = type and d = device

Table A-1. Parameter Types and Devices

TYPES:	A	H	P	S	F
DEVICES:					
A - P	x	x	x	x	
X		x	x	x	
Y		x	x	x	
Z		x	x	x	
I					x
D					d

x = valid, d = default

Valid Parameters

Except for the F type, the default device is the the current default drive.

Table A-2. Parameter Types

A	controls location of ASSEMBLER source file
H	controls location of HEX file
P	controls location of PRINT file
S	controls location of SYMBOL file
F	controls type of hex output FORMAT

Table A-3. Device Types

A - P	Drives A - P
X	console device
Y	printer device
Z	byte bucket
I	Intel hex format
D	Digital Research hex format

Table A-4. Invocation Examples

ASM86 IO	Assemble file IO.A86, produce IO.HEX IO.LST and IO.SYM.
ASM86 IO.ASM $ AD SZ	Assemble file IO.ASM on device D, produce IO.LST and IO.HEX, no symbol file.
ASM86 IO $ PY SX	Assemble file IO.A86, produce IO.HEX, route listing directly to printer, output symbols on console.
ASM86 IO $ FD	Produce Digital Research hex format.
ASM86 IO $ FI	Produce Intel hex format.

APPENDIX B

MNEMONIC DIFFERENCES FROM THE INTEL ASSEMBLER

The CP/M 8086 assembler uses the same instruction mnemonics as the INTEL 8086 assembler except for explicitly specifying far and short jumps, calls and returns. The following table shows the four differences:

Table B-1. Mnemonic Differences

Mnemonic Function	CP/M	INTEL
Intra segment short jump:	JMPS	JMP
Inter segment jump:	JMPF	JMP
Inter segment return:	RETF	RET
Inter segment call:	CALLF	CALL

APPENDIX C

ASM-86 HEXADECIMAL OUTPUT FORMAT

At the user's option, ASM-86 produces machine code in either Intel or Digital Research hexadecimal format. The Intel format is identical to the format defined by Intel for the 8086. The Digital Research format is nearly identical to the Intel format, but adds segment information to hexadecimal records. Output of either format can be input to the GENCMD, but the Digital Research format automatically provides segment identification. A segment is the smallest unit of a program that can be relocated.

Table C-1 defines the sequence and contents of bytes in a hexadecimal record. Each hexadecimal record has one of the four formats shown in Table C-2. An example of a hexadecimal record is shown below.

```
Byte number=>  0 1 2 3 4 5 6 7 8 9 ..............n

   Contents=>  : l l a a a a t t d d d ......... c c CR LF
```

Table C-1. Hexadecimal Record Contents

Byte	Contents	Symbol
0	record mark	:
1-2	record length	l l
3-6	load address	a a a a
7-8	record type	t t
9-(n-1)	data bytes	d d.....d
n-(n+1)	check sum	c c
n+2	carriage return	CR
n+3	line feed	LF

Table C-2. Hexadecimal Record Formats

Record type	Content	Format
00	Data record	: ll aaaa DT <data...> cc
01	End-of-file	: 00 0000 01 FF
02	Extended address mark	: 02 0000 ST ssss cc
03	Start address	: 04 0000 03 ssss iiii cc

```
ll    => record length - number of data bytes
cc    => check sum  - sum of all record bytes
aaaa  => 16 bit address
ssss  => 16 bit segment value
iiii  => offset value of start address
DT    => data record type
ST    => segment address record type
```

It is in the definition of record type (DT and ST) that Digital Research's hexadecimal format differs from Intel's. Intel defines one value each for the data record type and the segment address type. Digital Research identifies each record with the segment that contains it, as shown in Table C-3.

Table C-3. Segment Record Types

Symbol	Intel's Value	Digital's Value	Meaning
DT	00		for data belonging to all 8086 segments
		81H	for data belonging to the CODE segment
		82H	for data belonging to the DATA segment
		83H	for data belonging to the STACK segment
		84H	for data belonging to the EXTRA segment
ST	02		for all segment address records
		85H	for a CODE absolute segment address
		86H	for a DATA segment address
		87H	for a STACK segment address
		88H	for a EXTRA segment address

APPENDIX D

RESERVED WORDS

Table D-1. Reserved Words

Predefined Numbers

BYTE	WORD	DWORD

Operators

EQ	GE	GT	LE	LT
NE	OR	AND	MOD	NOT
PTR	SEG	SHL	SHR	XOR
LAST	TYPE	LENGTH	OFFSET	

Assembler Directives

DB	DD	DW	IF	RS
RB	RW	END	ENDM	EQU
ORG	CSEG	DSEG	ESEG	SSEG
EJECT	ENDIF	TITLE	LIST	NOLIST
INCLUDE	SIMFORM	PAGESIZE	CODEMACRO	PAGEWIDTH

Code-macro directives

DB	DD	DW	DBIT	RELB
RELW	MODRM	SEGFIX	NOSEGFIX	

8086 Registers

AH	AL	AX	BH	BL
BP	BX	CH	CL	CS
CX	DH	DI	DL	DS
DX	ES	SI	SP	SS

Instruction Mnemonics - See Appendix E.

APPENDIX E

ASM-86 INSTRUCTION SUMMARY

Table E-1. ASM-86 Instruction Summary

Mnemonic	Description	Section
AAA	ASCII adjust for Addition	10.3
AAD	ASCII adjust for Division	10.3
AAM	ASCII adjust for Multiplication	10.3
AAS	ASCII adjust for Subtraction	10.3
ADC	Add with Carry	10.3
ADD	Add	10.3
AND	And	10.3
CALL	Call (intra segment)	10.5
CALLF	Call (inter segment)	10.5
CBW	Convert Byte to Word	10.3
CLC	Clear Carry	10.6
CLD	Clear Direction	10.6
CLI	Clear Interrupt	10.6
CMC	Complement Carry	10.6
CMP	Compare	10.3
CMPS	Compare Byte or Word (of string)	10.4
CWD	Convert Word to Double Word	10.3
DAA	Decimal Adjust for Addition	10.3
DAS	Decimal Adjust for Subtraction	10.3
DEC	Decrement	10.3
DIV	Divide	10.3
ESC	Escape	10.6
HLT	Halt	10.6
IDIV	Integer Divide	10.3
IMUL	Integer Multiply	10.3
IN	Input Byte or Word	10.2
INC	Increment	10.3
INT	Interrupt	10.5
INTO	Interrupt on Overflow	10.5
IRET	Interrupt Return	10.5
JA	Jump on Above	10.5
JAE	Jump on Above or Equal	10.5
JB	Jump on Below	10.5
JBE	Jump on Below or Equal	10.5
JC	Jump on Carry	10.5
JCXZ	Jump on CX Zero	10.5
JE	Jump on Equal	10.5
JG	Jump on Greater	10.5
JGE	Jump on Greater or Equal	10.5
JL	Jump on Less	10.5
JLE	Jump on Less or Equal	10.5

Table E-1. (continued)

Mnemonic	Description	Section
JMP	Jump (intra segment)	10.5
JMPF	Jump (inter segment)	10.5
JMPS	Jump (8 bit displacement)	10.5
JNA	Jump on Not Above	10.5
JNAE	Jump on Not Above or Equal	10.5
JNB	Jump on Not Below	10.5
JNBE	Jump on Not Below or Equal	10.5
JNC	Jump on Not Carry	10.5
JNE	Jump on Not Equal	10.5
JNG	Jump on Not Greater	10.5
JNGE	Jump on Not Greater or Equal	10.5
JNL	Jump on Not Less	10.5
JNLE	Jump on Not Less or Equal	10.5
JNO	Jump on Not Overflow	10.5
JNP	Jump on Not Parity	10.5
JNS	Jump on Not Sign	10.5
JNZ	Jump on Not Zero	10.5
JO	Jump on Overflow	10.5
JP	Jump on Parity	10.5
JPE	Jump on Parity Even	10.5
JPO	Jump on Parity Odd	10.5
JS	Jump on Sign	10.5
JZ	Jump on Zero	10.5
LAHF	Load AH with Flags	10.2
LDS	Load Pointer into DS	10.2
LEA	Load Effective Address	10.2
LES	Load Pointer into ES	10.2
LOCK	Lock Bus	10.6
LODS	Load Byte or Word (of string)	10.4
LOOP	Loop	10.5
LOOPE	Loop While Equal	10.5
LOOPNE	Loop While Not Equal	10.5
LOOPNZ	Loop While Not Zero	10.5
LOOPZ	Loop While Zero	10.5
MOV	Move	10.2
MOVS	Move Byte or Word (of string)	10.4
MUL	Multiply	10.3
NEG	Negate	10.3
NOT	Not	10.3
OR	Or	10.3
OUT	Output Byte or Word	10.2

Table E-1. (continued)

Mnemonic	Description	Section
POP	Pop	10.2
POPF	Pop Flags	10.2
PUSH	Push	10.2
PUSHF	Push Flags	10.2
RCL	Rotate through Carry Left	10.3
RCR	Rotate through Carry Right	10.3
REP	Repeat	10.4
RET	Return (intra segment)	10.5
RETF	Return (inter segment)	10.5
ROL	Rotate Left	10.3
ROR	Rotate Right	10.3
SAHF	Store AH into Flags	10.2
SAL	Shift Arithmetic Left	10.3
SAR	Shift Arithmetic Right	10.3
SBB	Subtract with Borrow	10.3
SCAS	Scan Byte or Word (of string)	10.4
SHL	Shift Left	10.3
SHR	Shift Right	10.3
STC	Set Carry	10.6
STD	Set Direction	10.6
STI	Set Interrupt	10.6
STOS	Store Byte or Word (of string)	10.4
SUB	Subtract	10.3
TEST	Test	10.3
WAIT	Wait	10.6
XCHG	Exchange	10.2
XLAT	Translate	10.2
XOR	Exclusive Or	10.3

APPENDIX F

SAMPLE PROGRAM

Listing F-1. Sample Program APPF.A86

```
CP/M ASM86 1.09   SOURCE: APPF.A86        Terminal Input/Output
PAGE   1



                        title 'Terminal Input/Output'
                        pagesize 50
                        pagewidth 79
                        simform
                        ;
                        ;****** Terminal I/O subroutines ********
                        ;
                        ;       The following subroutines
                        ;       are included:
                        ;
                        ;       CONSTAT   -  console status
                        ;       CONIN     -  console input
                        ;       CONOUT    -  console output
                        ;
                        ;       Each routine requires CONSOLE NUMBER
                        ;       in the BL - register
                        ;
                        ;
                        ;       *****************
                        ;       *  Jump table:  *
                        ;       ****************
                        ;
                        CSEG            ; start of code segment
                        ;
                        jmp_tab:
0000 E90600                     jmp     constat
0003 E91900                     jmp     conin
0006 E92B00                     jmp     conout
                        ;
                        ;
                        ;       ***********************
                        ;       *  I/O port numbers   *
                        ;       ***********************
                        ;
```

```
                     ;
                     ;          Terminal 1:
                     ;
 0010                instatl          equ     10h       ; input status port
 0011                indatal          equ     11h       ; input port
 0011                outdatal         equ     11h       ; output port
 0001                readyinmaskl     equ     01h       ; input ready mask
 0002                readyoutmaskl    equ     02h       ; output ready mask
                     ;
                     ;          Terminal 2:
                     ;
 0012                instat2          equ     12h       ; input status port
 0013                indata2          equ     13h       ; input port
 0013                outdata2         equ     13h       ; output port
 0004                readyinmask2     equ     04h       ; input ready mask
 0008                readyoutmask2    equ     08h       ; output ready mask
                     ;
                     ;
                     ;          ***********
                     ;          * CONSTAT *
                     ;          ***********
                     ;
                     ;          Entry: BL - reg = terminal no
                     ;          Exit:  AL - reg = 0 if not ready
                     ;                            0ffh if ready
                     ;
                     constat:
0009 53E83F00                   push bx ! call okterminal
                     constatl:
000D 52                         push dx
000E B600                       mov  dh,0                ; read status port
0010 8A17                       mov  dl,instatustab [BX]
0012 EC                         in   al,dx
0013 224706                     and  al,readyinmasktab [bx]
0016 7402                       jz   constatout
0018 B0FF                       mov  al,0ffh
```

CP/M ASM86 1.09   SOURCE: APPF.A86   Terminal Input/Output

```
                     constatout:
001A 5A5B0AC0C3              pop dx ! pop bx ! or al,al ! ret
                     ;
                     ;
                     ;       *********
                     ;       * CONIN *
                     ;       *********
                     ;
                     ;       Entry: BL - reg = terminal no
                     ;       Exit:  AL - reg = read character
                     ;
001F 53E82900        conin:  push bx ! call okterminal !
0023 E8E7FF          coninl: call constatl           ; test status
0026 74FB                    jz   coninl
0028 52                      push dx                 ; read character
0029 B600                    mov  dh,0
002B 8A5702                  mov  dl,indatatab [BX]
002E EC                      in   al,dx
002F 247F                    and  al,7fh             ; strip parity bit
0031 5A5BC3                  pop dx ! pop bx ! ret
                     ;
                     ;
                     ;       **********
                     ;       * CONOUT *
                     ;       **********
                     ;
                     ;       Entry:  BL - reg = terminal no
                     ;               AL - reg = character to print
                     ;
0034 53E81400        conout: push bx ! call okterminal
0038 52                      push dx
0039 50                      push ax
003A B600                    mov  dh,0               ; test status
003C 8A17                    mov  dl,instatustab [BX]
                     conoutl:
003E EC                      in   al,dx
```



```
003F 224708                  and  al,readyoutmasktab [BX]
0042 74FA                    jz   conoutl
```

```
0044 58                  pop  ax                  ; write byte
0045 8A5704              mov  dl,outdatatab [BX]
0048 EE                  out  dx,al
0049 5A5BC3              pop dx ! pop bx ! ret
                 ;
                 ;
                 ;       ++++++++++++++
                 ;       + OKTERMINAL +
                 ;       ++++++++++++++
                 ;
                 ;       Entry:  BL - reg = terminal no
                 ;
                 okterminal:
004C 0ADB                or   bl,bl
004E 740A                jz   error
0050 80FB03              cmp  bl,length instatustab + 1
0053 7305                jae  error
0055 FECB                dec  bl
0057 B700                mov  bh,0
0059 C3                  ret
                 ;
005A 5B5BC3      error:  pop bx ! pop bx ! ret     ; do nothing
                 ;
                 ;************** end of code segment ***************
                 ;
                 ;       ****************
                 ;       * Data segment *
                 ;       ****************
                 ;
                         dseg

                 ;       **************************
                 ;       * Data for each terminal *
                 ;       **************************
```



```
                 ;

0000 1012        instatustab      db    instat1,instat2
0002 1113        indatatab        db    indata1,indata2
0004 1113        outdatatab       db    outdata1,outdata2
0006 0104        readyinmasktab   db    readyinmask1,readyinmask2
0008 0208        readyoutmasktab  db    readyoutmask1,readyoutmask2
```

```
;
;*************** end of file *********************
end

END OF ASSEMBLY. NUMBER OF ERRORS:   0
```

APPENDIX G

CODE-MACRO DEFINITION SYNTAX

```
<codemacro> ::= CODEMACRO <name> [<formal$list>]
                [<list$of$macro$directives>]
                ENDM

<name> ::= IDENTIFIER

<formal$list> ::= <parameter$descr>[{,<parameter$descr>}]

<parameter$descr> ::= <form$name>:<specifier$letter>
                      <modifier$letter>[(<range>)]

<specifier$letter> ::= A | C | D | E | M | R | S | X

<modifier$letter> ::= b | w | d | sb

<range> ::= <single$range>|<double$range>

<single$range> ::= REGISTER | NUMBERB

<double$range> ::= NUMBERB,NUMBERB | NUMBERB,REGISTER |
                   REGISTER,NUMBERB | REGISTER,REGISTER

<list$of$macro$directives> ::= <macro$directive>
                               {<macro$directive>}

<macro$directive> ::= <db> | <dw> | <dd> | <segfix> |
                      <nosegfix> | <modrm> | <relb> |
                      <relw> | <dbit>

<db> ::= DB NUMBERB | DB <form$name>

<dw> ::= DW NUMBERW | DW <form$name>

<dd> ::= DD <form$name>

<segfix> ::= SEGFIX <form$name>

<nosegfix> ::= NOSEGFIX <form$name>

<modrm> ::= MODRM NUMBER7,<form$name> |
            MODRM <form$name>,<form$name>

<relb> ::= RELB <form$name>

<relw> ::= RELW <form$name>

<dbit> ::= DBIT <field$descr>{,<field$descr>}
```

```
<field$descr> ::= NUMBER15 ( NUMBERB ) |
                  NUMBER15 ( <form$name> ( NUMBERB ) )

<form$name> ::= IDENTIFIER
```

NUMBERB is 8-bits
NUMBERW is 16-bits
NUMBER7 are the values 0, 1,. . , 7
NUMBER15 are the values 0, 1,. . , 15

APPENDIX H

ASM-86 ERROR MESSAGES

There are two types of error messages produced by ASM-86: fatal errors and diagnostics. Fatal errors occur when ASM-86 is unable to continue assembling. Diagnostics messages report problems with the syntax and semanics of the program being assembled. The following messages indicate fatal errors encountered by ASM-86 during assembly:

NO FILE
DISK FULL
DIRECTORY FULL
DISK READ ERROR
CANNOT CLOSE
SYMBOL TABLE OVERFLOW
PARAMETER ERROR

ASM-86 reports semantic and syntax errors by placing a numbered ASCII message in front of the erroneous source line. If there is more than one error in the line, only the first one is reported. Table H-1 summarizes ASM-86 diagnostic error messages.

Table H-1. ASM-86 Diagnostic Error Messages

Number	Meaning
0	ILLEGAL FIRST ITEM
1	MISSING PSEUDO INSTRUCTION
2	ILLEGAL PSEUDO INSTRUCTION
3	DOUBLE DEFINED VARIABLE
4	DOUBLE DEFINED LABEL
5	UNDEFINED INSTRUCTION
6	GARBAGE AT END OF LINE - IGNORED
7	OPERAND(S) MISMATCH INSTRUCTION
8	ILLEGAL INSTRUCTION OPERANDS
9	MISSING INSTRUCTION
10	UNDEFINED ELEMENT OF EXPRESSION
11	ILLEGAL PSEUDO OPERAND
12	NESTED "IF" ILLEGAL - "IF" IGNORED
13	ILLEGAL "IF" OPERAND - "IF" IGNORED
14	NO MATCHING "IF" FOR "ENDIF"
15	SYMBOL ILLEGALLY FORWARD REFERENCED - NEGLECTED
16	DOUBLE DEFINED SYMBOL - TREATED AS UNDEFINED
17	INSTRUCTION NOT IN CODE SEGMENT
18	FILE NAME SYNTAX ERROR
19	NESTED INCLUDE NOT ALLOWED
20	ILLEGAL EXPRESSION ELEMENT
21	MISSING TYPE INFORMATION IN OPERAND(S)
22	LABEL OUT OF RANGE
23	MISSING SEGMENT INFORMATION IN OPERAND
24	ERROR IN CODEMACROBUILDING

APPENDIX I

DDT-86 ERROR MESSAGES

Table I-1. DDT-86 Error Messages

Error Message	Meaning
AMBIGUOUS OPERAND	An attempt was made to assemble a command with an ambiguous operand. Precede the operand with the prefix "BYTE" or "WORD".
CANNOT CLOSE	The disk file written by a W command cannot be closed.
DISK READ ERROR	The disk file specified in an R command could not be read properly.
DISK WRITE ERROR	A disk write operation could not be successfully performed during a W command, probably due to a full disk.
INSUFFICIENT MEMORY	There is not enough memory to load the file specified in an R or E command.
MEMORY REQUEST DENIED	A request for memory during an R command could not be fulfilled. Up to eight blocks of memory may be allocated at a given time.
NO FILE	The file specified in an R or E command could not be found on the disk.
NO SPACE	There is no space in the directory for the file being written by a W command.
VERIFY ERROR AT s:o	The value placed in memory by a Fill, Set, Move, or Assemble command could not be read back correctly, indicating bad RAM or attempting to write to ROM or non-existent memory at the indicated location.

APPENDIX J

TMP LISTING

```
                    ;*****************************************************
                    ;*
                    ;*       Terminal Message Process
                    ;*
                    ;*       The TMP determines the user interface to MPM.
                    ;*       Much of the interface is available though
                    ;*       system calls.  This TMP takes advantage of
                    ;*       as much as possible for simplicity.  The TMP
                    ;*       could, for instance,  be easily modified to
                    ;*       force logins and have non-standard defaults.
                    ;*
                    ;*       With a little more work, The TMP could do all
                    ;*       command parsing and File Loading instead of
                    ;*       using the CLI COMMAND FUNCTION.  This is also
                    ;*       the place to AUTOLOAD programs for specific
                    ;*       users.  Suggestions are given in the MP/M-86
                    ;*       SYSTEM'S GUIDE.
                    ;*
                    ;*****************************************************

00FF                true            equ     0ffh
0000                false           equ     0
0000                unknown         equ     0
00E0                mpmint          equ     224          ; int vec for mpm

000D                cr              equ     13
000A                lf              equ     10

0002                mpm_conout      equ     2
0009                mpm_conwrite    equ     9
000A                mpm_conread     equ     10
000E                mpm_diskselect  equ     14
0019                mpm_getdefdisk  equ     25
0020                mpm_usercode    equ     32
0092                mpm_conattach   equ     146
0093                mpm_condetach   equ     147
0094                mpm_setdefcon   equ     148
0096                mpm_clicmd      equ     150
0098                mpm_parse       equ     152
00A0                mpm_setdeflst   equ     160
00A4                mpm_getdeflst   equ     164

0000                ps_run          equ     00          ; on ready list root
0001                pf_sys          equ     001h        ; system process
0002                pf_keep         equ     002h        ; do not terminate

0040                s_mpmseg    equ  word ptr 40H       ;begin MPM segment
004B                s_sysdisk   equ  byte ptr 04bh      ;system disk
0047                s_ncns      equ  byte ptr 47H       ;sys. consoles
0078                s_version   equ  word ptr 78h       ;ofst ver. str in SUP
```

```
0000                rsp_top       equ   0
0008                rsp_md        equ   008h
0010                rsp_pd        equ   010h
0040                rsp_uda       equ   040h
0140                rsp_bottom    equ   140h

0003                e_no_memory          equ   3    ; cant find memory
000C                e_no_pd              equ   12   ; no free pd's
000F                e_q_full             equ   15   ; full queue
0017                e_illdisk            equ   23   ; illegal disk #
0018                e_badfname           equ   24   ; illegal filename
0019                e_badftype           equ   25   ; illegal filetype
001C                e_bad_load           equ   28   ; bad ret. from BDOS load
001D                e_bad_read           equ   29   ; bad ret. from BDOS read
001E                e_bad_open           equ   30   ; bad ret. from BDOS open
001F                e_nullcmd            equ   31   ; null command sent
0025                e_ill_lst            equ   37   ; illegal list device
0026                e_ill_passwd         equ   38   ; illegal password


                    ;*****************************************************
                    ;*
                    ;*      TMP Shared Code and Constant Area
                    ;*
                    ;*****************************************************

                            cseg
                            org     0

                    ;===
                    mpm:    ; INTERFACE ROUTINE FOR SYSTEM ENTRY POINTS
                    ;===

0000 CDE0C3                 int mpmint ! ret

                    ;===
                    tmp:    ; PROGRAM MAIN - INITIALIZATION
                    ;===
                                    ; set default console # = TMP#
0003 8A160400E8AD           mov dl,defconsole ! call setconsole
     02

                                    ; set default disk = drive A
000A 1E8E1E0000             push ds ! mov ds,sysdatseg
000F 8A164B001F             mov dl,.s_sysdisk ! pop ds
0014 E8A502                 call setdisk

                                    ; set default user # = console
0017 8A160400E882           mov dl,defconsole ! call setuser
     02

                                    ; print version
001E E8AF02                 call attach
0021 1E8E1E0000             push ds ! mov ds,sysdatseg
```

```
0026 8B167800           mov dx,.s_version
002A 8E1E4000           mov ds,.s_mpmseg
002E E881021F           call print_ds_string ! pop ds
0032 E8A002             call detach

                                ; THIS IS WHERE A LOGIN ROUTINE MIGHT
                                ; BE IMPLEMENTED.  THE DATA FILE THAT
                                ; CONTAINS THE USER NAME AND PASSWORD
                                ; MIGHT ALSO CONTAIN AN INITIAL DEFAULT
                                ; DISK AND USER NUMBER FOR THAT USER.

                   ;===========
                   nextcommand:         ; LOOP FOREVER
                   ;===========

                                ; attach console
0035 E89802                 call attach

                                ; print CR,LF if we just sent command
0038 803E61020074           cmp cmdsent,false ! je noclearline
     08
003F C606610200                 mov cmdsent,false
0044 E85E02                     call crlf
                   noclearline:

                                ; set up and print user prompt
                                ; get current default user # and disk
                                ; this call should be made on every
                                ; loop in case the last command
                                ; has changed the default.

0047 B20DE84D02             mov dl,cr ! call prchar
004C E84F02                 call getuser
004F 8AD3E83302             mov dl,bl ! call prnum
0054 E86A02                 call getdisk
0057 B24102D3               mov dl,'A' ! add dl,bl
005B E83B02                 call prchar
005E BADF02                 mov dx,offset prompt
0061 E84402                 call print_string

                                ; Read Command from Console
0064 BADE01E87002           mov dx,offset read_buf ! call conread

                                ; echo newline
006A B20AE82A02             mov dl,lf ! call prchar

                                ; make sure not a null command
006F 8D1EE001               lea bx,clicb_cmd
0073 803EDF010074           cmp read_blen,0 ! je gonextcmd
     27
007A 803F3B7422                 cmp byte ptr [bx],';' ! je gonextcmd

                                      ; see if disk change
                                      ; if 'X:' change def disk to X
007F 803EDF010275                 cmp read_blen,2 ! jne clicall
```

```
     1E
0086 807F013A                         cmp byte ptr 1[bx],':'
008A 7518                             jne clicall

                                                ; change default disk
008C 8A17                                  mov dl,[bx]      ;get disk name
008E 80E25F                                and dl,5fh       ;Upper Case
0091 80EA41                                sub dl,'A'       ;disk number

                                                ; check bounds
0094 80FA007208                            cmp dl,0 ! jb gonextcmd
0099 80FA0F7703                            cmp dl,15 ! ja gonextcmd

                                                ; select default disk
009E E81B02                                    call setdisk
00A1 E991FF         gonextcmd:  jmp nextcommand

                    ;=======
                    clicall:          ; SEND CLI COMMAND
                    ;=======
                                            ; put null at end of input
00A4 BBE001                           mov bx,offset clicb_cmd
00A7 A0DF01B400                       mov al,read_blen ! mov ah,0
00AC 03D8C60700                       add bx,ax ! mov byte ptr [bx],0

                                            ; copy command string for err
                                            ; reporting later and to check
                                            ; for built in commands...
00B1 B94000                           mov cx,64
00B4 BEE001                           mov si,offset clicb_cmd
00B7 BF8802                           mov di,offset savebuf
00BA 1E07                             push ds ! pop es
00BC F3A5                             rep movsw

                                            ; parse front to see if
                                            ; built in command
00BE BE6802                           mov si,offset fcb
00C1 BF8802                           mov di,offset savebuf
00C4 E87601                           call parsefilename
00C7 E310                             jcxz goodparse
00C9 2BDB8A1EDF01                         sub bx,bx ! mov bl,read_blen
00CF 81C38802                             add bx,offset savebuf
00D3 C60724                               mov byte ptr [bx],'$'
00D6 E9E300                               jmp clierror


00D9 891E6202       goodparse:        mov parseret,bx
00DD 83FB007508                       cmp bx,0 ! jne haveatail
00E2 8A1EDF01                             mov bl,read_blen
00E6 81C38802                             add bx,offset savebuf
00EA C6072443       haveatail:        mov byte ptr [bx],'$' ! inc bx
00EE 803E68020074                     cmp fcb,0 ! je try_builtin
     03
00F5 E9A900                               jmp not_builtin
                                          ; is it USER command?
```

```
00F8 BE680246     try_builtin:     mov si,offset fcb ! inc si
00FC BFB703                        mov di,offset usercmd
00FF 0E07                          push cs ! pop es
0101 B90400F3A7                    mov cx,4 ! repz cmpsw
0106 7545                          jnz notuser
0108 BE6802                            mov si,offset fcb
010B 8B3E6202                          mov di,parseret
010F 83FF007425                        cmp di,0 ! je pruser
0114 47                                    inc di
0115 E82501                                call parsefilename
0118 83F900751C                            cmp cx,0 ! jne pruser
011D BE6802                                  mov si,offset fcb
0120 46                                      inc si
0121 8B14                                    mov dx,[si]
0123 E82701                                  call a_to_b
0126 80FB0F7708                              cmp bl,15 ! ja usererr
012B 8AD3                                      mov dl,bl
012D E87001                                    call setuser
0130 E90600                                    jmp pruser
0133 BA4D03       usererr:                   mov dx,offset usererrmsg
0136 E86F01                                  call printstring
0139 BA6E03       pruser:              mov dx,offset usermsg
013C E86901                            call printstring
013F E85C01                            call getuser
0142 8AD3E84001                        mov dl,bl ! call prnum
0147 E85B01                            call crlf
014A E9E8FE                            jmp nextcommand
                  notuser:
014D BE680246                      mov si,offset fcb ! inc si
0151 BFBF03                        mov di,offset printercmd
0154 0E07                          push cs ! pop es
0156 B90400F3A7                    mov cx,4 ! repz cmpsw
015B 7544                          jnz notprinter
015D BE6802                            mov si,offset fcb
0160 8B3E6202                          mov di,parseret
0164 83FF007424                        cmp di,0 ! je prprinter
0169 47                                    inc di
016A E8D000                                call parsefilename
016D 83F900751B                            cmp cx,0 ! jne prprinter
0172 BE6802                                  mov si,offset fcb
0175 46                                      inc si
0176 8B14                                    mov dx,[si]
0178 E8D200                                  call a_to_b
017B 80FBFF                                  cmp bl,0ffh
017E 7407                                    je printererr
0180 8AD3                                      mov dl,bl
0182 E84101                                    call setlist
0185 E306                                      jcxz prprinter
0187 BA7F03       printererr:                mov dx,offset printemsg
018A E81B01                                  call printstring
018D BAA303       prprinter:           mov dx,offset printermsg
0190 E81501                            call printstring
0193 E83501                            call getlist
0196 8AD3E8EC00                        mov dl,bl ! call prnum
```

```
019B E80701                                     call crlf
019E E994FE                                     jmp nextcommand
                    notprinter:
                    not_builtin:
                                                ; initialize Cli Control Block

01A1 C606DF0100                             mov clicb_net,0
                                                ; make cli call

01A6 C6066102FF                             mov cmdsent,true
01AB 8D16DF01B196                           lea dx,clicb ! mov cl,mpm_clicmd
01B1 E84CFE                                 call mpm
01B4 83FB007503                             cmp bx,0 ! jne clierror
01B9 E979FE                                     jmp nextcommand

                    ;========
                    clierror:
                    ;========
                    ; Cli call unsuccesful, analyze and display err msg
                    ;       input: CX = ERROR CODE

                                        ;null command?
01BC 83F91F7508               cmp cx,e_nullcmd ! jne not_nullcmd
01C1 C606610200                   mov cmdsent,false
01C6 E96CFE                       jmp nextcommand
                    not_nullcmd:
                                        ;no memory?
01C9 83F9037506               cmp cx,e_no_memory ! jne memory_ok
01CE BAE402E94E00                 mov dx,offset memerr ! jmp showerr
                    memory_ok:
                                        ;no pd in table?
01D4 83F90C7506               cmp cx,e_no_pd ! jne pd_ok
01D9 BAF702E94300                 mov dx,offset pderr ! jmp showerr
                    pd_ok:
                                        ;bad file spec?
01DF 83F918740F               cmp cx,e_badfname   ! je fname_bad
01E4 83F917740A               cmp cx,e_illdisk    ! je fname_bad
01E9 83F9267405               cmp cx,e_ill_passwd ! je fname_bad
01EE 83F9197506               cmp cx,e_badftype   ! jne fname_ok
01F3 BA0603E92900 fname_bad:  mov dx,offset fnameerr ! jmp showerr
                    fname_ok:
                                        ;bad load?
01F9 83F91C7405               cmp cx,e_bad_load ! je load_bad
01FE 83F91D7506               cmp cx,e_bad_read ! jne load_ok
0203 BA1503E91900 load_bad:   mov dx,offset loaderr ! jmp showerr
                    load_ok:
                                        ;bad open?
0209 83F91E7506               cmp cx,e_bad_open ! jne open_ok
020E BA2103E90E00                 mov dx,offset openerr ! jmp showerr
                    open_ok:
                                        ;RSP que full?
0214 83F90F7506               cmp cx,e_q_full ! jne que_ok
0219 BA3703E90300                 mov dx,offset qfullerr ! jmp showerr
                    que_ok:
                                        ;some other error...
```

```
021F BA3503                 mov dx,offset catcherr
                            ;jmp showerr

                  showerr:            ; Print Error String
                                      ; input: DX = address of Error
                                      ;             string in CSEG

0222 52                     push dx
0223 BA8802E88900           mov dx,offset savebuf ! call print_ds_string
0229 B23AE86B00             mov dl,':' ! call prchar
022E B220E86600             mov dl,' ' ! call prchar
0233 5A                     pop dx
0234 E87100E86B00           call printstring ! call crlf
023A E9F8FD                 jmp nextcommand

                  parsefilename:      ; SI = fcb   DI = string
023D B99800                           mov cx,mpm_parse
0240 BB6402                           mov bx,offset pcb
0243 893F897702                       mov [bx],di ! mov 2[bx],si
0248 8BD3E9B3FD                       mov dx,bx ! jmp mpm
                  a_to_b:             ;dl = 1st char, dh = 2nd char
024D 80FE207504                       cmp dh,' ' ! jne atob2char
0252 8AF2B230                            mov dh,dl ! mov dl,'0'
0256 80FE307229   atob2char:          cmp dh,'0' ! jb atoberr
025B 80FE397724                       cmp dh,'9' ! ja atoberr
0260 80FA30721F                       cmp dl,'0' ! jb atoberr
0265 80FA39771A                       cmp dl,'9' ! ja atoberr
026A 80EE3080EA30                        sub dh,'0' ! sub dl,'0'
0270 B800008AC2                          mov ax,0 ! mov al,dl
0275 52B10A                              push dx ! mov cl,10
0278 F6E15A                              mul cl ! pop dx
027B 8AD6B600                            mov dl,dh ! mov dh,0
027F 03C2                                add ax,dx
0281 8BD8C3                              mov bx,ax ! ret
0284 B3FFC3       atoberr:            mov bl,0ffh ! ret
                  prnum:              ; dl = num (0-15)
0287 80FA0A720A                       cmp dl,10 ! jb prnum_one
028C 52                                  push dx
028D B231E80700                          mov dl,'1' ! call prchar
0292 5A80EA0A                            pop dx ! sub dl,10
0296 80C230       prnum_one:          add dl,'0'
                                      ; jmp prchar
0299 B102E962FD   prchar:             mov cl,mpm_conout ! jmp mpm
029E B2FF         getuser:            mov dl,0ffh
02A0 B120E95BFD   setuser:            mov cl,mpm_usercode ! jmp mpm
02A5 BAE102       crlf:               mov dx,offset crlfstr
                                      ;jmp printstring
02A8 1E8CC88ED8   printstring:        push ds ! mov ax,cs ! mov ds,ax
02AD E802001FC3                       call print_ds_string ! pop ds ! ret
02B2 B109E949FD   print_ds_string:mov cl,mpm_conwrite ! jmp mpm
02B7 B194E944FD   setconsole:         mov cl,mpm_setdefcon ! jmp mpm
02BC B10EE93FFD   setdisk:            mov cl,mpm_diskselect ! jmp mpm
02C1 B119E93AFD   getdisk:            mov cl,mpm_getdefdisk ! jmp mpm
02C6 B1A0E935FD   setlist:            mov cl,mpm_setdeflst ! jmp mpm
02CB B1A4E930FD   getlist:            mov cl,mpm_getdeflst ! jmp mpm
```

```
02D0 B192E92BFD   attach:        mov cl,mpm_conattach ! jmp mpm
02D5 B193E926FD   detach:        mov cl,mpm_condetach ! jmp mpm
02DA B10AE921FD   conread:       mov cl,mpm_conread ! jmp mpm

                  ;*****************************************************
                  ;*
                  ;*      CONSTANTS (IN SHARED CODE SEGMENT)
                  ;*
                  ;*****************************************************

02DF 3E24         prompt         db      '>$'
02E1 0D0A24       crlfstr        db      13,10,'$'
02E4 3F4E6F742045 memerr         db      '?Not Enough Memory$'
     6E6F75676820
     4D656D6F7279
     24
02F7 3F5044205461 pderr          db      '?PD Table Full$'
     626C65204675
     6C6C24
0306 3F4261642046 fnameerr       db      '?Bad File Spec$'
     696C65205370
     656324
0315 3F4C6F616420 loaderr        db      '?Load Error$'
     4572726F7224
0321 3F43616E2774 openerr        db      '?Can''t Find Command$'
     2046696E6420
     436F6D6D616E
     6424
0335 3F24         catcherr       db      '?$'
0337 3F5253502043 qfullerr       db      '?RSP Command Que Full$'
     6F6D6D616E64
     205175652046
     756C6C24

034D 0D0A496E7661 usererrmsg     db      13,10,'Invalid User Number,'
     6C6964205573
     6572204E756D
     6265722C
0363 2049474E4F52                db      ' IGNORED',13,10,'$'
     45440D0A24
036E 0D0A55736572 usermsg        db      13,10,'User Number = $'
     204E756D6265
     72203D2024

037F 0D0A496E7661 printemsg      db      13,10,'Invalid Printer Number,'
     6C6964205072
     696E74657220
     4E756D626572
     2C
0398 2049474E4F52                db      ' IGNORED',13,10,'$'
     45440D0A24
03A3 0D0A5072696E printermsg     db      13,10,'Printer Number = $'
     746572204E75
     6D626572203D
     2024
```

```
 03B7 555345522020 usercmd        db      'USER    '
      2020
 03BF 5052494E5445 printercmd     db      'PRINTER '
      5220

                    ;*****************************************************
                    ;*
                    ;*      TMP Data Area - this area is copied once for
                    ;*              each system console.  The 'defconsole'
                    ;*              field is unique for each copy
                    ;*              - Each Data Area is run by a common
                    ;*              shared code segment.
                    ;*
                    ;*****************************************************

                        DSEG
                        org     rsp_top

 0000 0000          sysdatseg       dw      0
 0002 4700          sdatvar         dw      s_ncns
 0004 0000          defconsole      db      0,0
 0006 000000000000                  dw      0,0,0,0,0
      00000000

                        org     rsp_pd

 0010 00000000      pd              dw      0,0             ; link fields
 0014 00                            db      ps_run          ; status
 0015 C6                            db      198             ; priority
 0016 0300                          dw      pf_sys+pf_keep  ; flags
 0018 546D70202020                  db      'Tmp     '      ; Name
      2020
 0020 0400                          dw      offset uda/10h  ; uda seg
 0022 0000                          db      0,0             ; disk,user
 0024 0000                          db      0,0             ; ldisk,luser
 0026 FFFF                          dw      0ffffh          ; mem
 0028 00000000                      dw      0,0             ; dvract,wait
 002C 0000                          db      0,0             ; org,net
 002E 0000                          dw      0               ; parent
 0030 0000                          db      0,0             ; cns,abort
 0032 0000                          db      0,0             ; cin,cout
 0034 0000                          db      0,0             ; lst,sf3
 0036 0000                          db      0,0             ; sf4,sf5
 0038 00000000                      dw      0,0             ; reserved
 003C 00000000                      dw      0,0             ; pret,scratch

                        org     rsp_uda

 0040 000040010000 uda              dw      0,offset dma,0,0        ;0-7
      0000
 0048 000000000000                  dw      0,0,0,0                 ;8-fh
      0000
 0050 000000000000                  dw      0,0,0,0                 ;10-17
      0000
```

```
0058 000000000000              dw      0,0,0,0                   ;18-1f
     0000
0060 000000000000              dw      0,0,0,0                   ;20-27
     0000
0068 000000000000              dw      0,0,0,0                   ;28-2f
     0000
0070 00000000D801              dw      0,0,offset stack_top,0    ;30-37
     0000
0078 000000000000              dw      0,0,0,0                   ;38-3f
     0000
0080 000000000000              dw      0,0,0,0                   ;40-47
     0000
0088 000000000000              dw      0,0,0,0                   ;48-4f
     0000
0090 000000000000              dw      0,0,0,0                   ;50-57
     0000
0098 000000000000              dw      0,0,0,0                   ;58-5f
     0000
00A0 000000000000              dw      0,0,0,0                   ;60-67
     0000


                       org     rsp_bottom

0140              dma          rb      128

01C0 CCCCCCCCCCCC stack        dw      0cccch,0cccch,0cccch
01C6 CCCCCCCCCCCC              dw      0cccch,0cccch,0cccch
01CC CCCCCCCCCCCC              dw      0cccch,0cccch,0cccch
01D2 CCCCCCCCCCCC              dw      0cccch,0cccch,0cccch
01D8 0300         stack_top    dw      offset tmp      ; code offset
01DA 0000                      dw      unknown         ; code seg
01DC 0000                      dw      unknown         ; init. flags

  0080            maxcmdlen    equ     128

                       ; the Read Console Buffer and the
                       ; Cli Control Block share the same memory

01DE              read_buf     rb      0
01DE 80           read_maxcmd  db      128
01DF              clicb        rb      0
01DF              clicb_net    rb      0
01DF              read_blen    rb      1
01E0              clicb_cmd    rb      maxcmdlen + 1

0261 00           cmdsent      db      false

0262 0000         parseret     dw      0

0264 8802         pcb          dw      offset savebuf
0266 6802                      dw      offset fcb

0268              fcb          rb      32
0288              savebuf      rb      128
```

```
                                                  ;make sure hes is formed
 0308 00                            db      0
                            end
```

APPENDIX K

ECHO LISTING

```
                    ;
                    ;       ECHO - Resident System Process
                    ;       Print Command tail to console
                    ;

                    ;
                    ;       DEFININTIONS
                    ;

 00E0               mpmint          equ     224     ;mpm entry interrupt
 0009               mpm_conwrite    equ     9       ;print string
 0086               mpm_qmake       equ     134     ;create queue
 0087               mpm_qopen       equ     135     ;open queue
 0089               mpm_qread       equ     137     ;read queue
 008B               mpm_qwrite      equ     139     ;write queue
 0091               mpm_setprior    equ     145     ;set priority
 0093               mpm_condetach   equ     147     ;detach console
 0094               mpm_setdefcon   equ     148     ;set default console

 0030               pdlen           equ     48      ;length of Process
                                                    ;  Descriptor

 0020               p_cns           equ     byte ptr 020h   ;default cns
 0012               p_disk          equ     byte ptr 012h   ;default disk
 0013               p_user          equ     byte ptr 013h   ;default user
 0024               p_list          equ     byte ptr 024h   ;default list
 0000               ps_run          equ     0       ;PD run status
 0002               pf_keep         equ     2       ;PD nokill flag

 0000               rsp_top         equ     0       ;rsp offset
 0010               rsp_pd          equ     010h    ;PD offset
 0040               rsp_uda         equ     040h    ;UDA offset
 0140               rsp_bottom      equ     140h    ;end rsp header

                    ;
                    ;       CODE SEGMENT
                    ;
                            CSEG
                            org 0

0000 CDE0           mpm:    int mpmint
0002 C3                     ret

                    main:           ;create ECHO queue
0003 B186BAC301             mov cl,mpm_qmake ! mov dx,offset qd
0008 E8F5FF                 call mpm
                                    ;open ECHO queue
000B B187BA0903             mov cl,mpm_qopen ! mov dx,offset qpb
0010 E8EDFF                 call mpm
                                    ;set priority to normal
```

```
0013 B191BAC800                 mov cl,mpm_setprior ! mov dx,200
0018 E8E5FF                     call mpm

                                        ;ES points to SYSDAT
001B 8E060000                   mov es,sdatseg

                        loop:   ;forever
                                        ;read cmdtail from queue
001F B189BA0903                 mov cl,mpm_qread ! mov dx,offset qpb
0024 E8D9FF                     call mpm

                                        ;set default values from PD
0027 8B1E8302                   mov bx,pdadr
                        ;       mov dl,es:p_disk[bx]    ;p_disk=0-15
                        ;       inc dl ! mov disk,dl    ;make disk=1-16
                        ;       mov dl,es:p_user[bx]
                        ;       mov user,dl
                        ;       mov dl,es:p_list[bx]
                        ;       mov list,dl
002B 268A5720                   mov dl,es:p_cns[bx]
002F 88161903                   mov console,dl

                                        ;set default console
                        ;       mov dl,console
0033 B194E8C8FF                 mov cl,mpm_setdefcon ! call mpm

                                        ;scan cmdtail and look for '$' or 0.
                                        ;when found, replace w/ cr,lf,'$'

0038 8D1E8502B024               lea bx,cmdtail ! mov al,'$' ! mov ah,0
     B400
0040 8BD381C28300               mov dx,bx ! add dx,131
                        nextchar:
0046 3BDA770B                   cmp bx,dx ! ja endcmd
004A 38077407                   cmp [bx],al ! je endcmd
004E 38277403                   cmp [bx],ah ! je endcmd
0052 43EBF1                         inc bx ! jmps nextchar
                        endcmd:
0055 C6070D                     mov byte ptr [bx],13
0058 C647010A                   mov byte ptr 1[bx],10
005C C6470224                   mov byte ptr 2[bx],'$'

                                        ;write command tail
0060 8D168502B109               lea dx,cmdtail ! mov cl,mpm_conwrite
0066 E897FF                     call mpm
                                        ;detach console
0069 8A161903                   mov dl,console
006D B193E88EFF                 mov cl,mpm_condetach ! call mpm
                                        ;done, get next command
0072 EBAB                       jmps loop

                        ;
                        ;       DATA SEGMENT
                        ;
```

```
                                  DSEG
                                  org     rsp_top

0000 000000000000 sdatseg                 dw      0,0,0
0006 000000000000                         dw      0,0,0
000C 00000000                             dw      0,0

                                  org     rsp_pd

0010 00000000     pd                      dw      0,0               ; link,thread
0014 00                                   db      ps_run            ; status
0015 BE                                   db      190               ; priority
0016 0200                                 dw      pf_keep           ; flags
0018 4543484F2020                         db      'ECHO    '        ; name
     2020
0020 0400                                 dw      offset uda/10h    ; uda seg
0022 0000                                 db      0,0               ; disk,user
0024 0000                                 db      0,0               ; load dsk,usr
0026 0000                                 dw      0                 ; mem
0028 00000000                             dw      0,0               ; dvract,wait
002C 0000                                 db      0,0
002E 0000                                 dw      0
0030 00                                   db      0                 ; console
0031 000000                               db      0,0,0
0034 00                                   db      0                 ; list
0035 000000                               db      0,0,0
0038 000000000000                         dw      0,0,0,0
     0000

                                  org     rsp_uda

0040 0000DF010000 uda                     dw      0,offset dma,0,0          ;0
     0000
0048 000000000000                         dw      0,0,0,0
     0000
0050 000000000000                         dw      0,0,0,0                   ;10h
     0000
0058 000000000000                         dw      0,0,0,0
     0000
0060 000000000000                         dw      0,0,0,0                   ;20h
     0000
0068 000000000000                         dw      0,0,0,0
     0000
0070 000000007D02                         dw      0,0,offset stack_tos,0    ;30h
     0000
0078 000000000000                         dw      0,0,0,0
     0000
0080 000000000000                         dw      0,0,0,0                   ;40h
     0000
0088 000000000000                         dw      0,0,0,0
     0000
0090 000000000000                         dw      0,0,0,0                   ;50h
     0000
0098 000000000000                         dw      0,0,0,0
     0000
```

```
00A0 000000000000                   dw      0,0,0,0              ;60h
     0000

                         org        rsp_bottom

0140              qbuf              rb      131                  ;Queue buffer

01C3 0000         qd                dw      0                    ;link
01C5 0000                           db      0,0                  ;net,org
01C7 0000                           dw      0                    ;flags
01C9 4543484F2020                   db      'ECHO    '           ;name
     2020
01D1 8300                           dw      131                  ;msglen
01D3 0100                           dw      1                    ;nmsgs
01D5 00000000                       dw      0,0                  ;dq,nq
01D9 00000000                       dw      0,0                  ;msgcnt,msgout
01DD 4001                           dw      offset qbuf          ;buffer addr.

01DF              dma               rb      128

025F CCCCCCCCCCCC stack             dw      0cccch,0cccch,0cccch
0265 CCCCCCCCCCCC                   dw      0cccch,0cccch,0cccch
026B CCCCCCCCCCCC                   dw      0cccch,0cccch,0cccch
0271 CCCCCCCCCCCC                   dw      0cccch,0cccch,0cccch
0277 CCCCCCCCCCCC                   dw      0cccch,0cccch,0cccch
027D 0300         stack_tos         dw      offset main          ; start offset
027F 0000                           dw      0                    ; start seg
0281 0000                           dw      0                    ; init flags

0283              pdadr             rw      1                    ; QPB Buffer
0285              cmdtail           rb      129                  ; starts here
0306 0D0A24                         db      13,10,'$'

0309 0000         qpb               db      0,0                  ;must be zero
030B 0000                           dw      0                    ;queue ID
030D 0100                           dw      1                    ;nmsgs
030F 8302                           dw      offset pdadr         ;buffer addr.
0311 4543484F2020                   db      'ECHO    '           ;name to open
     2020

0319 00           console           db      0
                  ;disk             db      0
                  ;user             db      0
                  ;list             db      0

                         end
```

APPENDIX L

SYSTEM FUNCTION SUMMARY

Table L-1. System Function Summary

Number	Function Name	Input Parameters	Returned values
0	System Reset	none	none
1	Console Input	none	AL = char
2	Console Output	DL = char	none
3	Raw Console Input	none	AL = char
4	Raw Console Output	DL = char	none
5	List Output	DL = char	none
6	Direct Console I/O	see def	see def
7	Get I/O Byte	** Not supported	under MP/M-86 **
8	Set I/O Byte	** Not supported	under MP/M-86 **
9	Print String	DX = .Buffer	none
10	Read Console Buffer	DX = .Buffer	see def
11	Get Console Status	none	AL = 00/01
12	Return Version Number	none	AL= Version#
13	Reset Disk System	none	see def
14	Select Disk	DL = Disk Number	see def
15	Open File	DX = .FCB	AL = Dir Code
16	Close File	DX = .FCB	AL = Dir Code
17	Search for First	DX = .FCB	AL = Dir Code
18	Search for Next	none	AL = Dir Code
19	Delete File	DX = .FCB	AL = Dir Code
20	Read Sequential	DX = .FCB	AL = Err Code
21	Write Sequential	DX = .FCB	AL = Err Code
22	Make File	DX = .FCB	AL = Dir Code
23	Rename File	DX = .FCB	AL = Dir Code
24	Return Login Vector	none	AX = Login Vect*
25	Return Current Disk	none	AX = Cur Disk#
26	Set DMA Address	DX = .DMA	none
27	Get Addr(Alloc)	none	AX = .Alloc
28	Write Protect Disk	none	see def
29	Get R/O Vector	none	AX = R/O Vect*
30	Set File Attributes	DX = .FCB	see def
31	Get Addr(disk parms)	none	AX = .DPB
32	Set/Get User Code	see def	see def
33	Read Random	DX = .FCB	AL = Err Code
34	Write Random	DX = .FCB	AL = Err Code
35	Compute File Size	DX = .FCB	r0, r1, r2
36	Set Random Record	DX = .FCB	r0, r1, r2
37	Reset Drive	DX = drive Vect	AL = Err Code
38	Access Drive	DS = drive Vect	none
39	Free Drive	DS = drive Vect	none
40	Write Random w 0-fill	DS = .FCB	AL = Err Code
41	Test and Write Record	DS = .FCB	AL = Err Code
42	Lock Record	DS = .FCB (Current DMA Addr	AL = Err Code -> File ID)

Table L-1. (continued)

Number	Function Name	Input Parameters	Returned Values
43	Unlock Record	DX = .FCB (Current DMA ADDR	AL = Err Code -> File ID)
44	Set Multi-Sector Count	DL= # of Sectors	AL = Rtn Code
45	Set BDOS Error Mode	see def	none
46	Get Disk Free Space	DL = Disk #	see def
47	Chain To Program	see def	none
48	Flush Buffers	none	see def
50	Direct BIOS Call	DX = BD Addr.	AX = BIOS return
51	Set DMA Base	DX = DMA Seg.Addr	none
52	Get DMA Base	none	AX = DMA Offset
53	Get Max Mem	DX = MCB Addr	see def
54	Get Abs Max	DX = MCB Addr	see def
55	Alloc Mem	DX = MCB Addr	see def
56	Alloc Abs Max	DX = MCB Addr	see def
57	Free Mem	DX = MCB Addr	see def
58	Free All Mem	none	none
59	Program Load	DX = FCB Addr	AX = B.P.Seg
100	Set Directory Label	DX = .FCB	AL = Dir Code
101	Return Directory Label	DX = Disk #	AL = Label Data
102	Read File XFCB	DX = .XFCB	AL = Dir Code
103	Write File XFCB	DX = .XFCB	AL = Dir Code
104	Set Date and Time	DX = .TOD	none
105	Get Date and Time	DX = .TOD	none
106	Set Default Password	DX = .Password	none
107	Return Serial Number	DX = .serialnmb	serialnmb set
128	Absolute Memory Rqst	DX = .MD	AX = Err Code
129	Relocatable Mem Rqst	DX = .MD	AX = Err Code
130	Memory Free	DX = .MD	none
131	Poll	DL = Device	none
132	Flag Wait	DL = Flag	AX = Err Code
133	Flag Set	DL = Flag	AX = Err Code
134	Make Queue	DX = QD addr	none
135	Open Queue	DX = QPB Addr	AX = Err Code
136	Delete Queue	DX = QPB Addr	AX = Err Code
137	Read Queue	DX = QPB Addr	none
138	Conditional Read Queue	DX = QPB Addr	AX = Err Code
139	Write Queue	DX = QPB Addr	none
140	Conditional Write Queue	DX = QPB Addr	AX = Err Code
141	Delay	DX = #ticks	none
142	Dispatch	none	none
143	Terminate Process	DL = Term. Code	none
144	Create Process	DX = PD Addr	none
145	Set Priority	DL = Priority	none
146	Attach Console	none	none
147	Detach Console	none	none
148	Set Console	DL = Console	none
149	Assign Console	DX = ACB Addr	AX = Err Code
150	Send CLI Command	DX = CLBUF Addr	none
151	Call RPL	DX = CPB Addr	AX = result
152	Parse Filename	DX = PFCB Addr	see def
153	Get Console Number	none	AL = console #

Table L-1. (continued)

Number	Function	Input Parameters	Returned Values
154	System Data Address	none	AX = Sys Data Addr
155	Get Date and Time	DX = TOD Addr	none
156	Return PD Addr	none	AX = PD Addr
157	Abort Spec. Process	DX = ABP Addr	AL = Return Code
158	Attach List	none	none
159	Detach List	none	none
160	Set List	DL = List #	none
161	Cond. Attach List	none	AX = Err Code
162	Cond. Attach Console	none	AX = Err Code
163	MPM Version Number	none	AX = Version #
164	Get List Number	none	AL = list #

The following abbreviations are used in the table.

Addr = Address
Cond. = Conditional
Proc = Process
Rqst = Request
Spec. = Specified
term. = Terminate
char = ASCII character
Dir = Directory
Err = Error
Vect = Vector

Note: DL is the low-order half of register DX, and AL is the low-order half of register AX.

APPENDIX M

GLOSSARY

BCD: Acronym for Binary Coded Decimal. Representation of decimal numbers using binary digits. See Appendix N for binary representations of ASCII codes.

block: Basic unit of disk space allocation under MP/M-86. Each disk drive has a fixed block size (BLS) defined in its Disk Parameter Block in the XIOS. The block size can be 1K, 2K, 4K, 8K or 16K consecutive bytes. Blocks are numbered relative to zero so that each block is unique and has a byte displacement in a file of the Block Number times the Block Size.

boolean: Variable that can only have two values; usually interpreted as true/false , or on/off.

Checksum Vector (CSV): Contiguous data area in the XIOS with one byte for each directory sector to be checked, i.e. CKS bytes. A Checksum Vector is initialized and maintained for each logged-in drive. Each directory access by the system results in a checksum calculation which is compared with that in the Checksum Vector. If there is a discrepancy the drive is set to read-only status. This prevents the user from inadvertantly switching disks without logging-in the new disk. If not logged-in, the new disk is treated the same as the old one and data on it may be destroyed if writing is done.

CMD: File type for MP/M-86 command files. These are machine language object modules ready to be loaded and executed. Any file with this type may be executed by simply typing the file name after the drive prompt (e.g. 'A>'). For example, the program PIP.CMD may be executed by simply typing 'PIP'.

command: Set of instructions that are executed when the command name is typed after the system prompt. These instructions may be "built-in" the MP/M-86 system or may reside on disk as a file of type 'CMD. In general, MP/M-86 commands consist of three parts: the command name, the command tail, and a carriage return.

console: Primary I/O device used by MP/M-86. It usually consists of a CRT screen for displaying output and a keyboard for input.

control character: Non-printing ASCII character produced on the console by holding down the 'CTRL' (CONTROL) key while striking the character key (e.g. control-H means "hold down 'CTRL' and hit 'H'). Control characters are sometimes indicated using the up-arrow symbol (^), e.g. 'control-H' may be represented as '^H'. Certain control characters are treated as special commands by MP/M-86.

Default Buffer: 128-byte buffer maintained at 0080H in the Base Page. When the CLI loads a CMD file it initializes this buffer to the command tail, i.e. any characters typed after the CMD file name.

The first byte at 0080H contains the length of the command tail while the command tail itself begins at 0081H. A binary zero terminates the command tail. value. The 'I' command under DDT initializes this buffer in the same way as the CLI.

Default FCB: One of two FCBs maintained at 005CH and 006CH respectively, in the Base Page. The CLI function initializes the first default FCB from the first delimited field in the command tail and initializes the second default FCB from the next field in the command tail.

delimiters: ASCII characters used to separate constituent parts of a file specification. The CLI function recognizes certain delimiter characters as :.=;<>_', 'blank' and 'carriage return'. Several MP/M-86 commands also treat ,[]()$ as delimiter characters. It is advisable to avoid the use of delimiter characters and lower-case characters in filenames.

directory: Portion of a disk containing entries for each file on the disk and locations of the blocks allocated to the files. Each file directory element is in the form of a 32-byte FCB, although one file may have several elements depending on its size. The maximum number of directory elements supported is specified in the drive's Disk Parameter Block.

directory element: 32-byte element associated with each disk file. A file may have more than one directory element associated with it. There are four directory elements per directory sector. Directory elements may also be refered to as directory FCBs.

directory entry: File entry displayed when using the DIR command. This term may also be used to refer to a physical directory element (FCB).

disk: Magnetic media used for mass storage of data in the computer system. The term disk may refer to either a diskette, removable cartridge disk or fixed hard disk.

Disk Parameter Block (DPB): Table residing in the XIOS that defines the characteristics of a drive in the disk subsystem used with MP/M-86. The address of the DPB is in the Disk Parameter Header at DPbase + 0AH. Drives with the same characteristics may use the same Disk Parameter Header, and thus the same DPB. However drives with different characteristics must each have their own Disk Parameter Header and DPB's. The address of the drives Disk Parameter Header must be returned in registers HL when the BDOS calls the SELDSK entry point in the BIOS. BDOS Function 31 returns the DPB address.

Disk Parameter Header (DPH): 16-byte area in the XIOS containing information about the disk drive and a scratchpad area for certain BDOS operations. Given n disk drives, the Disk Parameter Headers are arranged in a table whose first row of 16 bytes corresponds to drive 0, with the last row corresponding to drive n-1.

extent (EX): 16K consecutive bytes in a file. Extents are numbered from 0 to 31. One extent may contain 1, 2, 4, 8 or 16 blocks. EX is the extent number field of a FCB and is a one byte field at FCB + 12, where FCB labels the first byte in the FCB. Depending on the Block Size (BLS) and the maximum data Block Number (DSM), a FCB may contain 1, 2, 4, 8 or 16 extents. The EX field is normally set to 0 by the user but contains the current extent number during file I/O. The term 'FCB Folding' is used to describe FCB's containing more than one extent. In CP/M version 1.4 each FCB contained only one extent. Users attempting to perform Random Record I/O and maintain CP/M 1.4 compatibility should be aware of the implications of this difference.

file: Collection of data containing from zero to 242,144 records. Each record contains 128 bytes and can contain either binary or ASCII data. ASCII data files consist of lines of data delineated by carriage return line feed sequences, meaning that one 128-byte record might contain one or more lines of text. Files consist of one or more extents, with 128 records per extent. Each file has one or more directory elements yet shows as only one directory entry when using the DIR command.

File Control Block (FCB): 36 consecutive bytes designated by the user for file I/O functions. The FCB fields are explained in Section 2.4. The term FCB is also used to refer a directory element in the directory portion of the allocated disk space. These contain the same first 32 bytes of the FCB, lacking only the Current Record and Random Record Number bytes.

HEX file format: Absolute output of ASM and MAC for the Intel 8080. A HEX file contains a sequence of absolute records which give a load address and byte values to be stored starting at the load address. (See Section 4.3).

I/O: Acronym for Input/Output operations or routines handling the input and output of data in the computer system.

logical drive: Logically distinct region of a physical drive. A physical drive may be divided into one or more logical drives, and designated with specific drive references (i.e., d:a or d:f, etc.). Thus at the user interface, it appears that there are several disks in the system.

Base Page: Memory region between 0000H and 0100H relative to the beginning of the data segment used to hold critical system parameters and which functions primarily as an interface region between user programs and the BDOS module. Note that in the 8080 Model, the code and data are intermixed in the code segment.

parse: Separate a command line into its constituent parts.

physical drive: Peripheral hardware device used for mass storage of data within the computer system.

read-only: Condition in which a drive may be read but not written to. A drive may be set to read-only status by using the SET or STAT utilities. The only other way a drive may be set to read-only status is if the checksum computed on a directory access does not match that stored in CSV when the drive is logged-in. This protects the user from switching disks without executing a disk reset. Files may also be set to read-only status with the Set or STAT utilities or the SET FILE ATTRIBUTES function (Function 30). Read-only is often abbreviated as "R/O".

record: Smallest unit of data in a disk file that can be read or written. A record consists of 128 consecutive bytes whose byte displacement in a file is the product of the Record Number times 128. A 128-byte record in a file occupies one 128-byte sector on the disk. If the blocking and deblocking algorithm is used several records may occupy each disk sector.

reentrant code: Code that can be used by one process while another is already executing it. Reentrant code must not be self-modifying; that is, it must be pure code and not contain data. The data for reentrant code can be kept in a separate data area or placed on the stack.

sector: 128 consecutive bytes in a disk file. A sector is the basic unit of data read and written on the disk by the XIOS. A sector can be one 128-byte record in a file or a sector of the directory. In some disk subsystems the disk sector size is larger than 128 bytes, usually a power of two such as 256, 512, 1024 or 2048 bytes. These disk sectors are refered to as Host Sectors. When the Host Sector size is larger than 128 bytes, Host Sectors must be buffered in memory and the 128-byte sectors must be blocked and deblocked from them.

spooling: Accumulating printer output in a file while the printer is kept busy printing so that programs with LIST output are not forced to wait until the printer is available.

source file: ASCII text file usually created with a text editor which is an input file to a system program, such as a language translator or text formattor.

stack: Reserved area of memory where the processor saves the return address when it receives a Call instruction. When the processor encounters a Return instruction, it restores the current address on the stack to the Instruction Pointer. Data such as the contents of the registers can also be saved on the stack. The Push instruction places data on the stack and the Pop instruction removes it. 8086 stacks are 16 bits wide; instructions operating on the stack add and remove stack items one word at a time. An item is pushed onto the stack by decrementing the stack pointer (SP) by 2 and writing the item at the SP address. In other words, the stack grows downward in memory.

track: Concentric ring on the disk; the standard IBM single density diskettes have 77 tracks. Each track consists of a fixed number of numbered sectors. Tracks are numbered from 0 to one less than the number of tracks on the disk. Data on the disk media is accessed by combinations of track and sector numbers.

user: Logically distinct subdivision of the directory. Each directory can be divided into 16 user numbers.

vector: Memory location used as an entry point into the operating system used for making system calls or interrupt handling.

wildcard: Filename containing either "?" or "*" characters. The BDOS directory search functions will match "?" with any single character and "*" with multiple characters.

APPENDIX N

ASCII AND HEXADECIMAL CONVERSIONS

This appendix contains tables of the ASCII symbols, including their binary, decimal, and hexadecimal conversions.

Table N-1. ASCII Symbols

Symbol	Meaning	Symbol	Meaning
ACK	acknowledge	FS	file separator
BEL	bell	GS	group separator
BS	backspace	HT	horizontal tabulation
CAN	cancel	LF	line feed
CR	carriage return	NAK	negative acknowledge
DC	device control	NUL	null
DEL	delete	RS	record separator
DLE	data link escape	SI	shift in
EM	end of medium	SO	shift out
ENQ	enquiry	SOH	start of heading
EOT	end of transmission	SP	space
ESC	escape	STX	start of text
ETB	end of transmission	SUB	substitute
ETX	end of text	SYN	synchronous idle
FF	form feed	US	unit separator
		VT	vertical tabulation

Table N-2. ASCII Conversion Table

Binary	Decimal	Hexadecimal	ASCII
0000000	000	00	NUL
0000001	001	01	SOH (CTRL-A)
0000010	002	02	STX (CTRL-B)
0000011	003	03	ETX (CTRL-C)
0000100	004	04	EOT (CTRL-D)
0000101	005	05	ENQ (CTRL-E)
0000110	006	06	ACK (CTRL-F)
0000111	007	07	BEL (CTRL-G)
0001000	008	08	BS
0001001	009	09	HT
0001010	010	0A	LF
0001011	011	0B	VT
0001100	012	0C	FF
0001101	013	0D	CR
0001110	014	0E	SO (CTRL-N)
0001111	015	0F	SI (CTRL-O)
0010000	016	10	DLE (CTRL-P)
0010001	017	11	DC1 (CTRL-Q)
0010010	018	12	DC2 (CTRL-R)
0010011	019	13	DC3 (CTRL-S)
0010100	020	14	DC4 (CTRL-T)
0010101	021	15	NAK (CTRL-U)
0010110	022	16	SYN (CTRL-V)
0010111	023	17	ETB (CTRL-W)
0011000	024	18	CAN (CTRL-X)
0011001	025	19	EM (CTRL-Y)
0011011	027	1B	ESC (CTRL-[)
0011100	028	1C	FS (CTRL-)
0011101	029	1D	GS (CTRL-])
0011110	030	1E	RS (CTRL-^)
0011111	031	1F	US (CTRL-_)
0100000	032	20	(SPACE)
0100001	033	21	!
0100010	034	22	"
0100011	035	23	#
0100100	036	24	$
0100101	037	25	%
0100110	038	26	&
0100111	039	27	'
0101000	040	28	(
0101001	041	29	)
0101010	042	2A	*
0101011	043	2B	+
0101100	044	2C	,
0101101	045	2D	-
0101110	046	2E	.
0101111	047	2F	/
0110000	048	30	0
0110001	049	31	1
0110010	050	32	2

Table N-2. (continued)

Binary	Decimal	Hexadecimal	ASCII
0110011	051	33	3
0110100	052	34	4
0110101	053	35	5
0110110	054	36	6
0110111	055	37	7
0111000	056	38	8
0111001	057	39	9
0111010	058	3A	:
0111011	059	3B	;
0111100	060	3C	<
0111101	061	3D	=
0111110	062	3E	>
0111111	063	3F	?
1000000	064	40	@
1000001	065	41	A
1000010	066	42	B
1000011	067	43	C
1000100	068	44	D
1000101	069	45	E
1000110	070	46	F
1000111	071	47	G
1001000	072	48	H
1001001	073	49	I
1001010	074	4A	J
1001011	075	4B	K
1001100	076	4C	L
1001101	077	4D	M
1001110	078	4E	N
1001111	079	4F	O
1010000	080	50	P
1010001	081	51	Q
1010010	082	52	R
1010011	083	53	S
1010100	084	54	T
1010101	085	55	U
1010110	086	56	V
1010111	087	57	W
1011000	088	58	X
1011001	089	59	Y
1011010	090	5A	Z
1011011	091	5B	[
1011100	092	5C	
1011101	093	5D	]
1011110	094	5E	^
1011111	095	5F	<
1100000	096	60	'
1100001	097	61	a
1100010	098	62	b
1100011	099	63	c
1100100	100	64	d

Table N-2. (continued)

Binary	Decimal	Hexadecimal	ASCII
1100101	101	65	e
1100110	102	66	f
1100111	103	67	g
1101000	104	68	h
1101001	105	69	i
1101010	106	6A	j
1101011	107	6B	k
1101100	108	6C	l
1101101	109	6D	m
1101110	110	6E	n
1101111	111	6F	o
1110000	112	70	p
1110001	113	71	q
1110010	114	72	r
1110011	115	73	s
1110100	116	74	t
1110101	117	75	u
1110110	118	76	v
1110111	119	77	w
1111000	120	78	x
1111001	121	79	y
1111010	122	7A	z
1111011	123	7B	{
1111100	124	7C	|
1111101	125	7D	}
1111110	126	7E	~
1111111	127	7F	DEL

INDEX

D

E

O

P

Q

R

S

T

U

V

W

X

MP/M-86 T.M. Operating System

PROGRAMMER'S GUIDE

Corrections to the First Printing - September 1981

Compiled October 5, 1981

PAGE 19

Figure 2-1. File Control Block Format

bytes 13 and 14 are labelled wrong.
change:

```
          ---------------
      ... |ex|s1|s2|rc| ...
          ---------------
           12 13 14 15
```

to:

```
          ---------------
      ... |ex|cs|rs|rc| ...
          ---------------
           12 13 14 15
```

PAGE 93

Function 20: READ SEQUENTIAL

in fiqure near end of page
change:

255 : Physical Error; refer to register H

to:

255 : Physical Error; refer to register AH
--

PAGE 112

Function 33: READ RANDOM

in fiqure near top of page
change:

255 : Physical Error; refer to register H

to:

255 : Physical Error; refer to register AH
--

PAGE 114

Function 34: WRITE RANDOM

in figure near top of page
change:

255 : Physical Error; refer to register H

to:

255 : Physical Error; refer to register <u>AH</u>

PAGE 176

Function 144: CREATE PROCESS

The figure near the top describing process priorities should be as follows:

```
          1 Initialization Process
  2 -    31 Interrupt handlers
 32 -    63 System Processes
 64 -   189 Undefined
        190 RSP Initialization
191 -   196 Undefined
        197 MPMSTAT
        198 Terminal Message Proccess
        199 Undefined
        200 Default Priority
201 -   254 User Processes
        255 Idle Process
```

SOFTWARE CHANGES IN ASM-86™ AND DDT-86™

ASM-86
------

1. Forward references in EQU's are flagged as errors.

2. A ! in a comment is ignored, comments extend to the physical end of line.

3. New directives: IFLIST and NOIFLIST are to control listing of false IF blocks.

4. IF directives may be nested to 5 levels.

5. New mnemonics implemented:
   a. JC, JNC
   b. CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, SCASB, SCASW, STOSB, STOSW

6. JNBE implemented correctly.

7. Segment override prefix is allowed in source operand of string instructions.

8. Relational operators in expressions return 0FFFFH if true.

9. Abort if invalid command tail encountered.

10. Abort if symbol table overflows.

11. Abort if disk or directory full.

12. Incomplete string flagged as error (no terminating quote).

13. Error reported if an invalid numeric quantity appears in EQU directive.

14. Source files are opened in RO mode for multiple access under MP/M-86.

15. Format of .LST file:
    a. form feed at start of file
    b. no form feed at end of file
    c. no cr,lf at top of each page
    d. fewer lines per page
    e. spaces between hex bytes deleted to allow more space for comments
    f. errors printed when NOLIST active
    g. absolute address field for relative instructions

16. Format of .SYM file:
    a. form feed at start of file
    b. symbols alphabetized within groups
    c. tabs expanded if symbols sent to printer ($SY)

17. Include files:
    a. file type defaults to .A86
    b. file type may have fewer than three characters
    c. abort if include file not found
    d. default to same drive as source when $a switch used

18. Programs with INCLUDE directives will assemble correctly under CP/M 1.4.

19. About 5.5K more space available for symbol table.

20. Use factor indicated at end of assembly (% usage of symbol table space).

21. Runs somewhat faster (especially with $PZ switch).

DDT-86

1. User programs default to CCP stack, rather than local stack in DDT86.

2. A command line starting with a ';' is treated as a comment.

3. Interrupts are disabled while a single instruction is being traced.

4. BDOS error mode is set to return BDOS errors for MP/M-86.

5. Files are closed after reading and loading for MP/M-86.

6. New Block Compare function implemented, with the same command form as the move function.