DIGITAL RESEARCH™

# Concurrent CP/M-86™ Operating System Programmer's Guide

# Concurrent CP/M-86™ Operating System Programmer's Guide

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

The <u>Concurrent CP/M-86 Operating System Programmer's Guide</u> was prepared using the Digital Research TEX$^{T.M.}$ Text Formatter and printed in the United States of America by Commercial Press/Monterey.

```
****************************************
*   First Edition:   September 1982    *
****************************************
```

# Foreword

Concurrent CP/M-86 T.M. is an operating system for the IBM Personal Computer. It supports four CP/M® programming environments called virtual consoles. A different task runs concurrently on each virtual console. The minimum hardware environment for Concurrent CP/M-86 includes an IBM Personal Computer with two disk drives and 256K bytes of Random Access Memory (RAM).

This manual describes the programming interface to Concurrent CP/M-86. Sections 1 through 4 describe the elements of the operating system, how Concurrent CP/M-86 monitors running processes, and all the system entry points.

Section 5 describes all the Concurrent CP/M-86 system function calls.

Section 6 contains an introduction to the Digital Research assembler ASM-86T.M. and the various options that can be used with it. Through one of these options, ASM-86 can generate 8086 machine code in either Intel® or Digital Research format. Appendix A describes these formats.

Section 7 discusses the elements of ASM-86 assembly language. It defines the ASM-86 character set, constants, variables, identifiers, operators, expressions, and statements.

Section 8 describes the ASM-86 housekeeping functions, such as conditional assembly, multiple source files inclusion, and control of the listing printout format.

Section 9 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 intrasegment short jump, intersegment jump, return, and call instructions. Appendix B summarizes these differences.

Section 10 discusses the Code-Macro facilities of ASM-86, including Code-Macro definition, specifiers, and modifiers, and nine special Code-Macro directives. This information is also summarized in Appendix G.

Section 11 discusses DDT-86T.M., the Dynamic Debugging Tool that allows the user to test and debug programs in the 8086 environment. The section includes a sample debugging session.

This manual is not a tutorial. Therefore, you should be familiar with the material covered in the IBM Personal Computer Guide To Operations and in the Concurrent CP/M-86 Operating System User's Guide.

# 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
# Concurrent CP/M-86 System Overview

## 1.1 Introduction

Concurrent CP/M-86 is a single-user, multitasking operating system that lets you run multiple programs simultaneously by dividing tasks between virtual consoles. Concurrent CP/M-86 is compatible with the CP/M-86 operating system. Applications programs have access to system functions used by Concurrent CP/M-86 to control the multiprogramming environment. As a result, Concurrent CP/M-86 supports extended features such as communication between and synchronization of independently running processes.

In the Concurrent CP/M-86 environment 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. It can be thought of as a logical machine that not only executes the program code but also executes code in the operating system.

When Concurrent CP/M-86 loads a program, it creates a process associated with the loaded program. Subsequently, it is the process rather than the program that controls all access to the system's resources. Concurrent CP/M-86 monitors the process, not the program. This distinction is a subtle one, but it is vital to your understanding of system operation.

Processes running under Concurrent CP/M-86 fall into two categories: transient processes and resident system processes. Transient processes run programs loaded into memory from disk. Resident System Processes run programs that are a part of the operating system itself. These programs are loaded during system initialization and usually perform operating system tasks.

For example, the Clock process is a predefined process that maintains the time of day within the operating system.

The following list summarizes the capabilities of Concurrent CP/M-86.

- Shared file system. This allows multiple programs access to common data files while maintaining data integrity.
- Virtual console handling. This lets a single user run multiple programs, each in its own console environment.
- Real-time process control. This allows communications and data acquisition without loss of information.

- Interprocess communication, synchronization, and mutual exclusion. These functions are provided by system queues.
- Logical interrupt mechanism using flags. This allows Concurrent CP/M-86 to interface with any physical interrupt structure.
- System timing functions. These functions enable processes running under Concurrent CP/M-86 to compute elapsed times, to delay execution for specified intervals, and to access and set the current date and time.

Functionally, Concurrent CP/M-86 is composed of several distinct modules:

- The Supervisor (SUP)
- The Real-Time Monitor (RTM)
- The Memory Management Module (MEM)
- The Character I/O Module (CIO)
- The Virtual Console Screen Manager
- The Basic Disk Operating System (BDOS)
- The Extended I/O System (XIOS)
- The Terminal Message Processor (TMP)

The SUP module handles miscellaneous system functions such as returning the version number or the address of the System Data Area. The SUP module 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 on demand from executing processes.

The CIO module handles all character I/O for console and list devices in the system.

The Virtual Console Screen Manager extends the CIO to support virtual console environments.

The BDOS is the hardware-independent module that contains the logically invariant portion of the file system for Concurrent CP/M-86. The BDOS file system is detailed in Section 2.

The XIOS is the hardware-dependent module that defines the Concurrent CP/M-86 interface to the IBM Personal Computer.

When Concurrent CP/M-86 is configured for a single virtual console and is executing a single program, its speed approximates that of CP/M-86. Where multiple processes are running on different virtual consoles, each individual process slows in proportion to the amount of I/O and compute resources it requires. 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 runs concurrently with other processes that are largely I/O bound. On the other hand, significant speed degradation occurs 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)

Concurrent CP/M-86 is controlled by a real-time multitasking nucleus, 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 performing these tasks can also be called by user programs.

### 1.3.1 Process Dispatching

Although Concurrent CP/M-86 is a multiprocessing operating system, at any given time only one process has access to the CPU resource. Unless it is specifically written to communicate or synchronize execution with other processes, the process runs unaware that other processes might 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 Concurrent CP/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, or suspended. A ready process is waiting for the CPU resource only. A running process is one that the CPU is currently executing. A suspended process is waiting for some other system resource or for 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, milliseconds, 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 Concurrent CP/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 detailed in the following sections.

Concurrent CP/M-86 is a priority system. This means that the Dispatcher selects the process with the best priority and gives it the CPU resource. Processes with the same priority are round-robin scheduled; they are given equal shares of the system's resources. With priority dispatching, control is never passed to a worse-priority process if there is a better-priority process on the Ready list. Because best priority, compute-bound processes tend to monopolize the CPU resource, it is advisable to reduce their priority to avoid degrading overall system performance.

Concurrent CP/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 Concurrent CP/M-86. They are used for communicating messages between processes, for synchronizing process execution, and for mutual exclusion. Like files, queues are made, opened, deleted, read from, and written to with appropriate system function calls.

Each system queue is composed of two parts: the queue descriptor, and the queue buffer. These are special data structures implemented in Concurrent CP/M-86 as memory files containing room for a specified number of fixed-length messages.

When a queue is created by the Make Queue function call, it is assigned an eight-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 conditionally or unconditionally. If no messages exist in the queue when a conditional read is performed, or if 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-served basis.

Mutual exclusion queues are a special type of queue under Concurrent CP/M-86. They contain one message of zero length and are typically assigned a name beginning with the upper-case letters MX. 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, 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, freeing the resource for other processes.

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

Mutual exclusion queues have one other feature different from other queues. When a process reads a message from a mutual exclusion queue, the Process Descriptor Address is noted within the queue description. This establishes the owner of the queue message. If the process is aborted while it owns the mutual exclusion message, the RTM automatically writes the message back to all mutual exclusion queues whose messages are owned by the aborted process. This grants other processes access to the protected resource.

### 1.3.3 System Timing Functions

Concurrent CP/M-86 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 to set the internal time and date. Subsequently, the Clock process makes another Flag Wait call and suspends itself until the flag is set again. Concurrent CP/M-86 provides functions for setting and accessing the internal date and time. The file system also 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 Management Module (MEM)

The Memory Management Module handles all memory management functions. Concurrent CP/M-86 supports an extended, fixed partition model of memory management. In practice, the exact method the operating system uses to allocate and free memory is transparent to the programmer. In fact, the programmer should write code that is independent of the Memory Management Module by using only the Concurrent CP/M-86 system functions as described in Section 5. If the system functions are not used, incompatibility can result. Future versions of Concurrent CP/M-86 might support different versions of the Memory Management Module, depending on the classes of memory management hardware available.

## 1.5 Basic Disk Operating System (BDOS)

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

- File locking. Files opened under Concurrent CP/M-86 cannot be opened or deleted by other tasks. This feature prevents accidental conflicts with other tasks.
- Shared access to files. As a special option, independent users can open the same file in shared or Unlocked mode. Concurrent CP/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 tasks.
- Date Stamps
- Password Protection
- Extended Error Modes

## 1.6 Character I/O Module (CIO)

The Character I/O module handles all console and list I/O. Under Concurrent CP/M-86, every character I/O device is associated with a data structure called a Console Control Block (CCB) or a List Control Block (LCB). The CCB contains the current owner, a linked list of Process Descriptors (PDs) waiting for access, line editing variables, and status information. CCBs and LCBs reside in the XIOS.

## 1.7 Virtual Console Screen Manager

The Virtual Console Screen Manager interfaces the Character I/O module to the Extended I/O module. This module accepts I/O requests from applications programs, handling the request internally with the PIN process or passing the request on to the XIOS. This scheme maps many virtual consoles onto one physical console.

The Screen Manager contains a single physical input process (PIN) and a virtual output process, VOUTxx. xx indicates the virtual console number for each virtual console. PIN accepts and interprets input from the physical console, handling Switch Console, CTRL-C, CTRL-S, and CTRL-Q functions itself. PIN also passes remaining characters on to the queue associated with the currently selected console, VINQxx. During output, the Character I/O module writes each character to the XIOS if in Dynamic mode or to the appropriate VOUTQxx if in the Buffered mode.

## 1.8 Extended Input/Output System (XIOS)

The XIOS module is similar to the CP/M-86 Basic Input/Output System (BIOS) module, but it is extended in several ways. Primitive functions such as console I/O are modified to support multiple virtual consoles. Several new primitive functions support Concurrent CP/M-86's additional features, including elimination of wait loops for real-time activities.

## 1.9 Terminal Message Processor (TMP)

The Concurrent CP/M-86 TMPs are resident system processes that accept command lines from the virtual consoles and call the Command Line Interpreter to execute them. The TMP prints the prompt on the virtual consoles.

Each virtual console has an independent TMP defining that console's environment, including default disk, user number, printer, and console.

## 1.10 Transient Programs

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

## 1.11 System Function Calling Conventions

When a Concurrent CP/M-86 process makes a system function call, it follows the protocol shown in Table 1-1.

**Table 1-1. Registers for System Function Calls**

| Entry Parameters | |
|---|---|
| Register | CL: Function Number<br>DL: Byte Parameter<br>or<br>DX: Word Parameter<br>or<br>DX: Address - Offset<br>DS: Address - Segment |
| **Return Values** | |
| Register | AL: Byte Return<br>or<br>AX: Word Return<br>or<br>AX: Address - Offset<br>ES: Address - Segment |
| | BX: Same as AX<br>CX: Error Code |

The contents of registers SI, DI, and BP are preserved through the operating system calls.

## 1.12 Error Handling

Most system functions return an error code to the calling process. In Concurrent CP/M-86, the CX register is reserved as the error code return register.

There is one set of error codes common to all functions except those in the BDOS module. BDOS functions have their own error codes, explained in Section 2.15. The error codes for the non-BDOS Concurrent CP/M-86 system functions are shown in Table 1-2.

**Table 1-2. Concurrent CP/M-86 Error Codes**

| Code# | Definition |
|---|---|
| 00H | NO ERROR |
| 01H | FUNCTION NOT IMPLEMENTED |
| 02H | ILLEGAL FUNCTION NUMBER |
| 03H | CANNOT FIND MEMORY |
| 04H | ILLEGAL SYSTEM FLAG NUMBER |
| 05H | FLAG OVERRUN |
| 06H | FLAG UNDERRUN |
| 07H | NO UNUSED QUEUE DESCRIPTORS LEFT IN QD TABLE |
| 08H | NO UNUSED QUEUE BUFFER AREA LEFT |
| 09H | CANNOT FIND QUEUE |
| 0AH | QUEUE IN USE |
| 0BH | QUEUE NOT ACTIVE |
| 0CH | NO UNUSED PROCESS DESCRIPTORS LEFT IN PD TABLE |
| 0DH | QUEUE ACCESS DENIED |
| 0EH | EMPTY QUEUE |
| 0FH | FULL QUEUE |
| 10H | CLI QUEUE MISSING |
| 11H | NO QUEUE BUFFER SPACE |
| 12H | NO UNUSED MEMORY DESCRIPTORS LEFT IN MD TABLE |
| 13H | ILLEGAL CONSOLE NUMBER |
| 14H | CANNOT FIND PD BY NAME |
| 15H | CONSOLE DOES NOT MATCH |
| 16H | NO CLI PROCESS |
| 17H | ILLEGAL DISK NUMBER |
| 18H | ILLEGAL FILE NAME |
| 19H | ILLEGAL FILE TYPE |
| 1AH | CHARACTER NOT READY |
| 1BH | ILLEGAL MEMORY DESCRIPTOR |
| 1CH | BAD LOAD |
| 1DH | BAD READ |
| 1EH | BAD OPEN |
| 1FH | NULL COMMAND |
| 20H | NOT OWNER |
| 21H | NO CODE SEGMENT IN LOAD FILE |
| 22H | ACTIVE PD |
| 23H | CANNOT TERMINATE |
| 24H | CANNOT ATTACH |
| 25H | ILLEGAL LIST DEVICE NUMBER |
| 26H | ILLEGAL PASSWORD |

End of Section 1

# Section 2
# The Concurrent CP/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 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, identified by user numbers 0 through 15. Only files belonging to the current user number are visible in the directory. For example, the Concurrent CP/M-86 DIR utility displays only 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 need not be concerned with directory and drive organization when using the file system functions.

An eight-character filename and a three-character filetype identify 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 filename and filetype. Multiple files can be specified by a wildcard specification. A wildcard specification uses one or more ? marks in the filename or filetype, indicating that any character can match that position. Thus, a filename and filetype consisting 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.

### 2.1.1 File Access Functions

The file access category includes functions to 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, 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. The function also updates the directory to permanently record the current status of the file.

### 2.1.2 Directory Functions

BDOS directory functions operate on existing file entries in a drive 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.

Search and Delete are the only BDOS functions that allow wildcard file specifications. All other directory and file-related functions require a specific file specification. 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.

### 2.1.3 Drive-Related Functions

BDOS drive-related functions 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 used, and the type of date and time stamping to be performed for files on the drive.

This category also includes 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 Concurrent CP/M-86, a successful drive reset operation must be performed on drives that support removable media before changing disks.

### 2.1.4 Miscellaneous Functions

Miscellaneous functions 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 BDOS file system operations:

- 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. Read these sections before attempting to use the system functions described individually in Section 5.

## 2.2 File Naming Conventions

Under Concurrent CP/M-86, file specifications consist of four parts: the drive specification (d), the filename, the filetype, and the file password. The general format for a command line file specification is shown below:

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

The drive specification shows the drive in which the file is located. The filename and filetype identify the file. The password is necessary only if password protection is enabled.

The drive specification, filetype, and password are optional. 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 you do not specify the drive, the current default drive is indicated. The filename contains one to eight nondelimiter characters; the filetype, one to three nondelimiter characters; and the password field, one to eight nondelimiter characters. All alphabetic characters must be in upper-case. The Parse Filename function pads all three parts of the filespec with blanks, if necessary. Omitting the optional filetype or password implies a file 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 |
|---|---|
| : | 03AH |
| . | 02EH |
| ; | 03BH |
| = | 03DH |
| , | 02CH |
| / | 02FH |
| [ | 05BH |
| ] | 05DH |
| < | 03CH |
| > | 03EH |

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

Avoid the characters "(" and ")" in the filename and filetype because they are commonly used delimiters. The characters * and ? must not be used in the filename and filetype except as wildcard specifications. If the Parse Filename function encounters a * in a filename or filetype, 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 match a ? in the filename or filetype to the corresponding position of any directory entry in the current user number. 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 filetype as an error.

It is not mandatory to follow the file naming conventions of Concurrent CP/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 filetype contains a lower-case letter.

As a general rule, the filetype names the generic category of a particular file. The filename distinguishes individual files within each category. The filetypes listed below name some of the established generic categories.

| | |
|---|---|
| ● A86 | 8086 Assembler Source |
| ● ASM | Assembler Source |
| ● BAK | ED Source Backup |
| ● BAS | Basic Source File |
| ● BRS | 8080 Banked RSP File |
| ● CMD | 8086 Command File |
| ● COM | 8080 Command File |
| ● CON | CCP/M-86 Modules |
| ● DAT | Data File |
| ● HEX | HEX Machine Code |
| ● H86 | ASM-86 HEX File |
| ● INT | Intermediate File |
| ● LIB | Library File |
| ● LST | List File |
| ● PLI | PL/I Source File |
| ● PRL | Page Relocatable |
| ● REL | Relocatable Module |
| ● RSP | Resident System Process |
| ● SPR | System Page Relocatable |
| ● SUB | SUBMIT File |
| ● SUP | Startup File |
| ● SYM | SYM Symbol File |
| ● SYS | System 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 shows 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 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 the files existing 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. Only files belonging to the current user number are visible in the directory.

Each disk file consists of a set of up to 262,144 128-byte records. Each record in a file is identified by its position in the file. This position is called the Random Record Number. If a file is created sequentially, the first record has a position of zero; 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 in 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. A sparse file containing two records, one at position zero, the other at position 262,143, would consume only two data blocks in the data area. Sparse files can only be created and accessed randomly, not sequentially.

**Note:** any data block allocated to a file is permanently allocated to the file until the file is deleted. The BDOS supports no other mechanism for releasing data blocks belonging to a file.

Source files under Concurrent CP/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 CTRL-Z (1AH) or a real end-of-file, returned by the BDOS read operation. CTRL-Z characters embedded in 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. These functions reference an FCB to determine the files to be operated on. Certain fields in the FCB also start 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 and Write File XFB, and Lock and Unlock Record functions, the BDOS file system maintains the current file state and position within the user's FCB. 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 register DX containing the offset, and DS containing 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 long. The FCB format is shown below.

| dr | f1 | f2 | ... | f8 | t1 | t2 | t3 | ex | cs | rs | 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 filetype in ASCII upper-case, with high bit = 0. t1', t2', and t3' denote the high bit of these positions and are file attribute bits.<br><br>t1' = 1 => Read-Only file,<br>t2' = 1 => System file,<br>t3' = 1 => File has been archived. |
| ex | contains the current extent number. This is usually set to 0 by the calling process, but it can range from 0 to 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 to 128. |
| d0...dn | filled in by Concurrent CP/M-86, reserved for system use. |
| cr | current record to read or write in a sequential file operation, set to zero by the calling process when a file is opened or created. |
| r0,r1,r2 | optional Random Record Number in the range 0-262,143 (0 - 3FFFFH). ro,r1,r2 constitute an 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 filename and filetype 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. 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 user should not modify the FCB. Open FCBs are checksum verified to protect the integrity of the file system. If a process modifies an open FCB, the next read, write, or close function call will return with a checksum error. (See Section 2.10.) 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 before 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, BDOS returns a checksum error. 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. 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, a process which creates or modifies files must close the files at the termination of any write processing. Otherwise, data might be lost.

As a general rule, 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. A process must close a file before other processes on the system can access the file.

The space in the system Lock list is limited. If a process attempts to open a file and no space exists in the system Lock list, or if the process exceeds the process open file limit, the BDOS denies the open operation and usually aborts the calling process.

The high-order bits of the FCB filename (f1',...,f8') and filetype (t1',t2',t3') 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 function within the file system as file attributes and interface attributes.

## 2.4.1 File Attributes

The file attributes f1',...,f4' and t1',t2',t3' indicate that a file has a defined attribute. These bits are recorded in a file's directory FCBs. File attributes can be set or reset only 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',and 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 Concurrent CP/M-86 system file. System files are not usually displayed by Concurrent CP/M-86 DIR utility. User-zero system files can also 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 an 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. An archive program can test this attribute in each of the file's directory FCBs using the BDOS Search For First and Search For Next functions. If all directory FCBs have the archive attribute set, the file has not been modified since the previous archive. Note that the Concurrent CP/M-86 PIP utility supports file archival.

## 2.4.2 Compatibility Attributes

Compatibility attributes allow certain programs developed under earlier Digital Research operating systems to run under the Concurrent CP/M-86, MP/M II$^{T.M.}$, or MP/M-86$^{T.M}$operating systems.

Concurrent CP/M-86, MP/M II, and MP/M-86 all share a similar file system enhanced to support a multitasking environment. Some of these enhancements are restrictions not present in the earlier CP/M type file systems found in CP/M-80$^{T.M.}$, CP/M-86$^{T.M.}$, or MP/M-80$^{T.M.}$. The compatibility attributes ease some of these restrictions for software developed under earlier file systems, allowing CP/M type software to run under operating systems implementing the enhanced file system. MP/M II was the first operating system in this family to incorporate the enhanced file system.

The compatibility attributes are not usually needed to run CP/M-86 software under Concurrent CP/M-86.

For example, one of these restrictions operates when a process opens a file in the Default (Locked) mode. Concurrent CP/M-86 does not allow other processes on the system to open, delete, or rename the file until the process opening the file either closes the file or terminates. Concurrent CP/M-86 does not allow a process to perform file operations with an FCB that has not been activated by a successful open or make operation or with an FCB that has been deactivated by a close operation.

The preceding example describes restrictions required to prevent collisions between independent processes during file access. Another new Concurrent CP/M-86 restriction sets limits on how a process can modify open FCBs. These limits are enforced by checksum verification of open FCBs. They protect the integrity of the Concurrent CP/M-86 file system from corrupted FCBs.

Software developed under the CP/M type file systems opens files in the Default (Locked) mode. Under Concurrent CP/M-86 these files are protected in case several programs try to write to them simultaneously. Concurrent CP/M-86 allows simultaneous file modification in an orderly way. (See Section 2.11.)

Note that the new Concurrent CP/M-86 restrictions do not protect one process or task from its own actions; rather, they ensure that one task cannot adversely affect other tasks on the system.

The new Concurrent CP/M-86 file system restrictions create little difficulty for new application development. In fact, they enforce good programming practice. However, because of these new restrictions, some CP/M and MP/M T.M. software written before MP/M II's release does not run on Concurrent CP/M-86. Multiple copies of some CP/M-86 programs do not run because the default open mode for Concurrent CP/M-86 is a locked mode in which only one process can open a file.

## Compatibility Attributes F1' Through F4'

To address these problems, Digital Research has added compatibility attributes to MP/M II, MP/M-86, and Concurrent CP/M-86. These compatibility attributes are defined as attributes F1' through F4' of program files. The Command Line Interpreter (CLI) interrogates these attributes during program loading and modifies the Concurrent CP/M-86 ground rules for the loaded program as described below. Concurrent CP/M-86, as supplied by Digital Research for the IBM Personal Computer, always sets the 'F3' compatiblity attribute.

**Note:** do not use these compatibility attributes with new software. Use these compatibility attributes with working software developed for CP/M-80, CP/M-86, and MP/M-80. Compatibility attribute F4' disables FCB checksum verification on read and write operations. Use this attribute sparingly and only with programs known to work.

**Table 2-3. Compatibility Attribute Definitions**

| Attribute | Definition |
|---|---|
| **F1'** | MP/M 1.1 Default Open. Processes running with this attribute have all files opened in locked mode marked as Read-Only in the System Lock List. All processes with this attribute set can read and write to common files with no restrictions. However, no record locking is provided. This attribute also allows a process to write to a file opened by another process in Read-Only mode. To be safe, make all static files such as program and help files Read-Only when this compatibility attribute is used. |
| **F2'** | Partial Close default. Processes running with this attribute have their default close mode changed from permanent close to partial close. This attribute is for programs that close a file to update the directory but continue to use the file. Note that Concurrent CP/M-86 assumes a process has finished with a file when the number of closes issued to the file equals the number of opens. A side effect of this attribute is that files opened by a process are not released until the process terminates. It might be necessary to set the System Lock List parameters to high values when using this attribute. |
| **F3'** | Ignore Close Checksum Errors. This attribute changes the way Close Checksum errors are handled for a process. Usually, a message is printed on the console and the process terminates. When this attribute is set and a checksum error is detected during a close operation, the file is closed if a Lock list item exists for the file. Otherwise, an unsuccessful close error code is returned to the calling process. Under Concurrent CP/M-86, the CLI function always sets this attribute when loading programs from disk to accommodate applications programs that require them. |
| **F4'** | Disable FCB Checksum verification for read and write operations. Setting this attribute also sets attributes F2' and F3'. This attribute should be used carefully because it effectively disables Concurrent CP/M-86's file security. Use this attribute only with software with which it is known to work. |

Procedure For Using The Compatibility Attributes

Use the Concurrent CP/M-86 utility SET to set the combination of compatibility attributes you want in the program name.

**Examples:**

```
0A>SET filespec [F1=on]
0A>SET filespec [F1=on,F3=on]
0A>SET filespec [F4=on]
```

If you have a program that runs under CP/M-80, CP/M-86, or MP/M-80 1.1 but does not run properly under Concurrent CP/M-86, use the following guidelines to select the compatibility attributes to set for the program.

- If the program ends with the message, File Currently Opened, when multiple copies of the program are run, set compatibility attribute F1', or place all common static files under User 0 with the SYS and R/O attributes set.
- If the program terminates with the message, Close Checksum Error, set compatibility attribute F3'.
- If the program terminates with an I/O error, try running the program with attribute F2' set. If the problem persists, then try attribute F4'. Use attribute F4' only as a last resort.

## 2.4.3 Interface Attributes F5' Through F8'

The interface attributes are f5' through f8'. These attributes cannot be used as file attributes. Interface attributes f5' and f6' 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 2-4 shows the f5' and f6' interface attribute definitions for these functions.

**Table 2-4. BDOS Interface Attributes**

| Function | Attribute |
|---|---|
| Open File | f5' = 1 : Open in Unlocked Mode<br>f6' = 1 : Open in Read-Only Mode |
| Make File | f5' = 1 : Open in Unlocked Mode<br>f6' = 1 : Assign password to file |
| Close File | f5' = 1 : Partial Close |
| Delete File | f5' = 1 : Delete file XFCBs only |

Section 5 details the interface attributes for each of the preceding functions. 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 reference (wildcard specifications). In general, a ? mark in the filename, filetype, 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. If specified, these functions return all directory entries on the disk regardless of user number and including empty entries. A directory FCB the first byte of which contains the value E5H is an empty directory entry.

## 2.5 User Number Conventions

The Concurrent CP/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. The current user number applies to all drives on the system. Furthermore, the FCB format does not contain any field that can 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 before issuing the BDOS function call for the file. Note that if a process attempts to read or write to a file under a user number different from 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. The sending process is usually the TMP. However, the sending process can 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 and the TMP regains control, the original user number for the console is restored.

User 0 has special properties under Concurrent CP/M-86. With some restrictions, the file system automatically opens a file under user zero, if the file is not present under the current user number. 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 convention allows utilities that can include overlays and any other commonly accessed files to be placed on user zero, but remain available for access by 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 Concurrent CP/M-86 manager control over user-zero files directly accessible from other user numbers.

## 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 be associated optionally 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 |
| type | - filetype |
| 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 whether passwords for password protected files are to be required, whether date and time stamping for files is to be performed, and whether 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. Directory label format is 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<br>bit 7 - require passwords for files<br>bit 6 - perform access time stamping<br>bit 5 - perform update time stamping<br>bit 4 - Make creates XFCBs<br>bit 0 - directory label exists<br>(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 disk 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, because file password protection is an option controlled by the directory label. Thus, access to the directory label password provides 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 can 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 2-5 shows the difference in access level allowed to BDOS functions when the password is <u>not</u> supplied.

**Table 2-5. 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 2-6 lists the BDOS functions that test for password.

**Table 2-6. 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, before making the function call.

**Note:** the BDOS maintains the assigned default password on a per process basis. Processes inherit their parent process's default password. You can set a given TMP's default password using Set. Programs loaded by this TMP inherit the same default password.

## 2.8 File Date and Time Stamps

The BDOS file system can record when a file was created, last accessed, and 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. 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 Concurrent CP/M-86 internal date and time when it records a date and time stamp. The Concurrent CP/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 for opening files. They are defined below.

### 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 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 Concurrent CP/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 usually the case. The BDOS record locking and unlocking functions are supported only 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 if 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 nonzero, 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 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, see 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 (t1') 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 2-7 shows the definition of the FCB interface attributes f5' and f6' for the BDOS Open File function.

**Table 2-7. 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 2-8 shows the definition of the FCB interface attribute f5' for the Make File function.

**Table 2-8. 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

Concurrent CP/M-86 because the BDOS file system maintains file allocation information in open FCBs in the user's memory region, and Concurrent CP/M-86 does not support memory protection. In the worst case, a program that crashes on Concurrent CP/M-86 can take down the entire system. Therefore, Concurrent CP/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 by

- performing checksum verification of open FCBs.
- monitoring 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 defines 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. See 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 Concurrent CP/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 is a special case because a process cannot control whether this situation occurs and it can impact more than one process. (See Section 2.13.)

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 Concurrent CP/M-86.

## Extended File Locking

Extended file locking enables a Concurrent CP/M-86 process to maintain a lock on a file even after the file is closed. This facility allows a process to rename, set the attributes, or delete a file without having to contend with interference from other processes after the file is closed.

A process can also reopen a file with an extended lock and continue regular file processing. For example, a process can open a file, perform file operations on the file, close the file, rename the file, reopen the file under its new name, and proceed with file operations, without ever losing the file's Lock list item and control over the file.

Extended file locking is available only to files that are opened in the default open mode (Locked mode). To extend a file's lock, set interface attribute F6' when closing the file. This attribute is interrogated by the Close function only when it is closing a file permanently. Thus, interface attribute F5' must be reset when the Close call is made. If a file has been opened N times (more than once), this attribute is interrogated only when the file is closed for the Nth time.

To maintain an extended file lock through a Rename File call or a Set File Attributes call, set interface attribute F5' of the referenced FCB when making the call. This attribute is honored only for extended file locks, not normal locks. Setting attribute F5' also maintains an extended file lock for the Delete File function, but setting this attribute also changes the nature of the Delete function to an XFCB-Only delete. If successful, all three of these functions delete a file's extended lock item with attribute F5' reset. If they return with an error code, the extended lock item is not deleted.

A standard open call can be made to resume file operations on a file with an extended lock. The Open mode, however, is restricted to the default Locked mode. You can use extended locks to

- open file EXLOCK.TST in Locked mode
- perform file operations on the file EXLOCK.TST using the open FCB
- close file EXLOCK.TST with interface attribute F6' set to retain the file's lock item
- use the Rename File function to change the name of the file to EXLOCK.NEW with interface attribute F5' set to retain the file's extended lock item
- open the file EXLOCK.NEW in Locked mode
- perform operations on the file EXLOCK.NEW using the opened FCB

- close file EXLOCK.NEW with interface attribute F6' set to retain the file's lock item
- set the Read-Only attribute and release the file's lock item by using the Set File Attributes function with interface attribute F5' reset. At this point, the file EXLOCK.NEW becomes available for access by another process.

## 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. The 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 Concurrent CP/M-86, the number of records a process can lock is restricted.

The Test And Write Record function, on the other hand, performs its verification at the I/O level. In a single 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 process's 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. 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 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 multisector 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.

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 AH.

## 2.13 Flushing Buffers in the XIOS

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; it is required for BDOS write operations. The reverse process is called deblocking; 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 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 an issue only for systems that implement blocking and deblocking because of the postponement of physical write operations. If the system crashes while the physical buffer is pending, data is 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 to reinitialize a drive directory for file operations.

When Concurrent CP/M-86 is initiated, 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, Concurrent CP/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 Concurrent CP/M-86, the drive reset operation is conditional. 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 removable. Concurrent CP/M-86 determines whether a drive is permanent or removable by interrogating a bit in the drive's Disk Parameter Block (DPB) in the XIOS. A high-order bit of 1 in the DPB checksum vector size field designates the drive as permanent. Under Concurrent CP/M-86, a drive's designation is critical to the reset operation described below.

The BDOS first determines whether 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 removable 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 performed only if a media change is detected on the drive. The logic flow of the drive reset operation is shown in Figure 2-4.

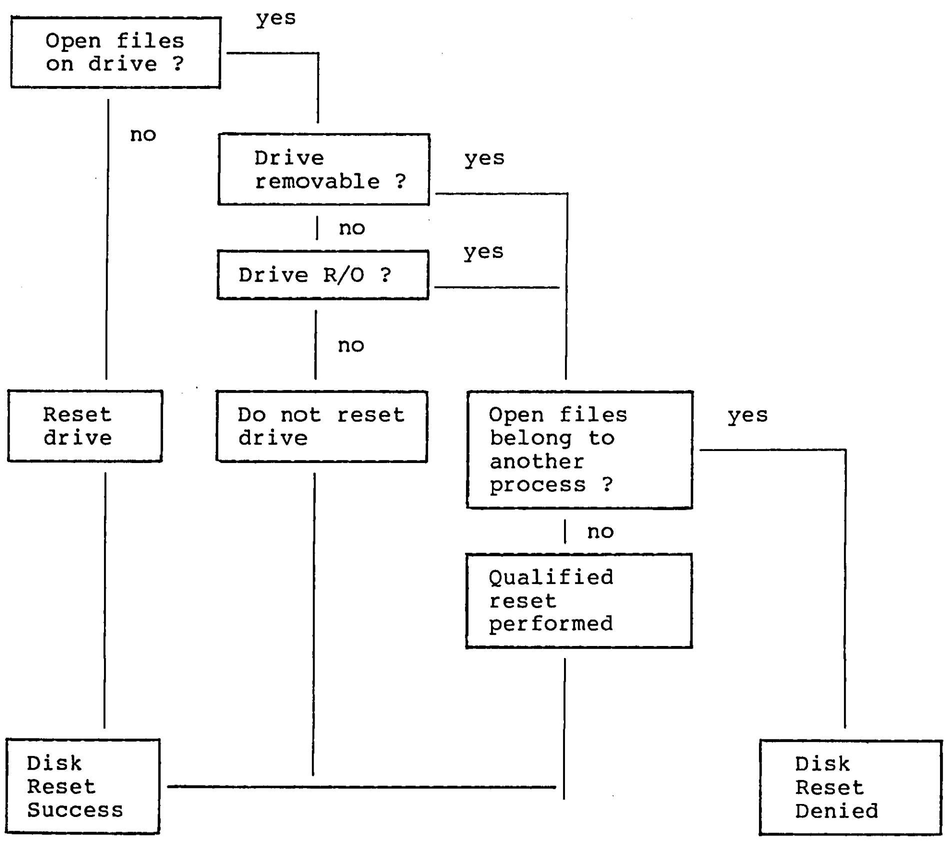


**Figure 2-4. Disk System Reset**

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 as FCB checksum error by the BDOS file system.

The Access Drive and Free Drive functions perform special actions under Concurrent CP/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 Concurrent CP/M-86. This function can be used to set the specified drive to Read-Only. However, Concurrent CP/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 removable 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 one of three ways:

- It can return to the calling process with return codes in AX register identifying the error.
- It can display an error message on the console and abort the process.
- 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 Error 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). The latter two return modes ensure that Concurrent CP/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 following tables detail BDOS physical and extended error messages.

**Table 2-9. BDOS Physical Errors**

| Error | Explanation |
|---|---|
| Bad Sector | 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, causing this error message. |
| Select | 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 before 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. |
| File R/O | The BDOS returns the File R/O error whenever a process makes a write operation to a file with the R/O attribute set. |

**Table 2-9. (continued)**

| Error | Explanation |
|---|---|
| R/O | 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. |

**Table 2-10. BDOS Extended Errors**

| Error | Explanation |
|---|---|
| File Opened in Read-Only Mode | 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 nonzero, 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. |
| File Currently Opened | 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. |
| Close Checksum Error | 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. |

**Table 2-10. (continued)**

| Error | Explanation |
|---|---|
| Password Error | The BDOS returns the File Password error when the file password is not supplied or when it is incorrect. |
| File Already Exists | 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 filetype. |
| Illegal ? in FCB | The BDOS returns the Illegal ? in FCB error whenever the BDOS detects a ? in the filename or filetype of the passed FCB for the BDOS Rename File, Set File Attributes, Open File, and Make File functions. |
| Open File Limit Exceeded | The BDOS returns the Open File Limit Exceeded error when a process exceeds the file lock limit specified in the system Lock list. The Open File, Make File, and Access Drive functions can return this error. |
| No Room in System Lock List | The BDOS returns the No Room in System Lock list error when no room for new entries exists within the system Lock list. 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 let programs written for CP/M-86 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

Table 2-11 lists error code definitions for register AL.

**Table 2-11. BDOS Error Codes**

| Code | Definition |
|---|---|
| 00H | Function successful |
| 01H | Reading unwritten data<br>No available directory space (Write Sequential) |
| 02H | No available data block |
| 03H | Cannot close current extent |
| 04H | Seek to unwritten extent |
| 05H | No available directory space |
| 06H | Random record number out of range |
| 07H | Record match error (Test and Write) |
| * 08H | Record locked by another process<br>(restricted to files opened in unlocked mode) |
| 09H | Invalid FCB (previous BDOS read or write call returned an error code and invalidated the FCB) |
| 0AH | FCB checksum error |
| * 0BH | Unlocked file unallocated block verify error |
| ** 0CH | Process record lock limit exceeded |
| ** 0DH | Invalid File ID |
| ** 0EH | No room in System Lock List |
| 0FFH | Physical error : refer to register AH |

* - 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 2-12.

**Table 2-12. BDOS Directory Codes**

| Code | Meaning |
|---|---|
| 00H - 03H | successful function |
| 0FFH | 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's 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 2-13.

**Table 2-13. BDOS Error Flags**

| Code | Meaning |
|---|---|
| 00H | successful function |
| 0FFH | 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 2-14 lists the physical and extended error codes returned in N2.

**Table 2-14. BDOS Physical and Extended Errors**

| Code | Meaning |
|---|---|
| 00H | - No error or not a register AH error |
| 01H | - Bad Sector : permanent error |
| 02H | - R/O : Read-Only Diskette |
| 03H | - R/O File : Read-Only file |
| | - File Opened in Read-Only Mode |
| 04H | - Select : drive select error |
| 05H | - File Currently Open |
| 06H | - Close Checksum Error |
| 07H | - Password Error |
| 08H | - File Already Exists |
| 09H | - Illegal ? in FCB |
| 0AH | - Open File Limit Exceeded |
| 0BH | - 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 0FFFFH. Otherwise, they return no error code.

Under Concurrent CP/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. If the BDOS is not in Return Error mode, these functions also display an error message identifying the process that prevented the requested operation.

## 2.16 Programming Guidelines

This discussion emphasizes those areas of Concurrent CP/M-86 where restrictions exist that did not exist in MP/M-80 or do not exist in CP/M-80 or CP/M-86.

Always follow this sequence when performing file operations requiring an open file. Under Concurrent CP/M-86, these operations are the BDOS read, write, lock, and unlock record commands.

- Activate a file's FCB with a BDOS Open or Make function call before using the FCB in a file operation. Verify that the Open or Make operation was successful. Concurrent CP/M-86 only accepts FCBs activated by a successful Open or Make call for open file operations. If an FCB that has not been activated is used, Concurrent CP/M-86 returns a checksum error.

- Perform all file operations using activated FCBs. Note that Concurrent CP/M-86 does not deactivate an activated FCB when it returns error codes for file operations. Generally, only the current record and random record fields of an activated FCB should be modified. In addition, all file operations with an activated FCB must be made under the user number that was in effect when the FCB was activated. A similar restriction applies to activated FCBs that specify the default drive. All file operations specifying such an FCB must be made under the current drive that was in effect when the FCB was activated. Item 3 in this list covers the complete rules regarding activated FCB modification.

- If a process has completed file operations on a file but still has a significant amount of processing left to do, the file should be closed. This applies even if the file was not modified. With some exceptions, the Lock list entry associated with a file in the system Lock list is not released until a file is permanently closed.

Concurrent CP/M-86 restricts access to a file by other processes while a Lock list item for the file resides in the system Lock list. It is not necessary to close input files if a process is about to end. At termination, all lock items belonging to a process are released. Output files, however, must always be closed or data might be lost.

Note that a successful permanent close operation deactivates the FCB and removes the file's item from the system Lock list. If the deactivated FCB is used in a subsequent open file operation, Concurrent CP/M-86 returns a checksum error.

If a process opens the same file more than once, a matching number of close commands must be issued to the file to remove the file's Lock list item from the system Lock list. Thus, if a file has been opened N times, the first N-1 close operations issued to the file default to partial close operations. Only the last close, close operation N, is interpreted as a permanent close. By definition, a permanent close is a close operation that removes the referenced file's item from the system Lock list. Note that only one Lock list item is allocated in the system Lock list for a file regardless of the number of FCBs a process has opened for the file.

The following list specifies how an activated FCB can be changed without affecting the FCB checksum. Concurrent CP/M-86 returns a checksum error code and does not perform the requested operation if an FCB with an invalid checksum is used in an open file operation.

- FCB(0) cannot specify a new drive.
- With the exception of interface attributes F5' and F6' for the BDOS Close function, FCB(1) through FCB(11) cannot be changed.

- The high-order 3 bits of FCB(12) cannot be changed. The low-order 5 bits can be changed. Note that when a file is opened in the default open mode (Locked mode), the high-order 3 bits of this FCB field are set to zeros.

- FCB(13) cannot be changed.
- FCB(14) and FCB(15) can be changed.
- FCB(16) through FCB(31) cannot be changed.
- FCB(32) through FCB(35) can be changed.

If compatibility with future releases of MP/M and CP/M is a requirement, programs should restrict open FCB modification to the FCB fields 32 through 35. In particular, Digital Research does not support techniques that involve modifying fields 12, 14, and 15 of open FCBs.

Processes that access a printer must issue a Detach List device to free the printer before another process can use the printer. If the Detach List call is not made, a process that accesses a printer continues to own the printer until it ends.

CP/M programs that make direct BIOS calls for disk I/O do not work under Concurrent CP/M-86. Concurrent CP/M-86 does support direct BIOS calls for the console and printer but not to the disk.

The following procedure is a protocol that multiple processes can use to coordinate record update and addition operations to a shared file. Each process must open the shared file in unlocked mode. This procedure also assumes that records containing binary zeros are null records.

- Attempt to lock the record.
- If the lock attempt fails because another process has locked the record, delay and repeat the procedure.
- If the lock attempt fails because the record does not exist in the file, add a record initialized to binary zeros to the file with the BDOS Write Random with Zero Fill command and repeat the procedure. Note that files opened in Unlocked mode are extended in block units and not in record units as is the case for files opened in the default Locked mode.
- If the lock attempt succeeds, read the record, update it, and then unlock it.

Multiple FCB I/O is a technique that involves opening each extent for a file independently and maintaining it in a table in memory. Then random I/O is handled by selecting the proper FCB from the table, setting the current record field to the proper record number within the extent, and making a sequential Read or Write command. When processing is completed, each FCB is closed. The maximum file size that can be accessed with this technique is 512K bytes. This limits the maximum table size to 32 FCBs. Note that this technique provides a method of performing random I/O that is compatible with CP/M 1.4.

Multiple FCB I/O must be performed carefully under Concurrent CP/M-86 because of the restrictions Concurrent CP/M-86 places on file operations to provide file security. Generally, an FCB should not be used in file I/O unless it has been activated and it should not be modified while it is activated. In addition, the number of opens and closes issued to a file is important. Note that all 32 bytes of each extent's FCB should be maintained in the open FCB table. Also, verify that interface attribute F8' is set to 1 in all FCBs if the first FCB has F8' set to 1. F8' set to 1 indicates the file was opened under user 0 although the current user number is nonzero. (See Function 15.)

End of Section 2

# Section 3
# Transient Commands

## 3.1 Transient Process Load and Exit

You can initiate a transient process by entering a command at a system console. The console's TMP (Terminal Message Processor) then calls the Command Line Interpreter function (See Function 150), 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. The CLI function calls the Parse Filename function that 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. Concurrent CP/M-86 divides the DMA address into 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 so 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 terminates a process by typing a single CTRL-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 as 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 |
|---|---|
| 01H | Code Group |
| 02H | Data Group |
| 03H | Extra Group |
| 04H | Stack Group |
| 05H | Auxiliary Group #1 |
| 06H | Auxiliary Group #2 |
| 07H | Auxiliary Group #3 |
| 08H | Auxiliary Group #4 |
| 09H | Shared Code Group |
| 0AH | Unused, but Reserved |
| 0BH | " |
| 0CH | " |
| 0DH | " |
| 0EH | " |
| 0FH | 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 080H, for example, the size of the group is 0800H (2048 decimal) bytes. |
| A-Base | defines the base paragraph address for a nonrelocatable 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, because 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 Concurrent CP/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.

|  | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
|  | L | M | H | L | H |  |
| 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 Concurrent CP/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. Concurrent CP/M-86 Base Page Values**

The various fields in 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 that might 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, for example, 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.

- 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.

**Note:** 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 Relationships

Under Concurrent CP/M-86, when one process creates another process, there is a parent/child relationship between them. 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 also inherits interrupt vectors (0-4 inclusive, 224 and 225) that the parent process initialized.

End of Section 3

# 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.

**Table 4-1. Concurrent CP/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 that 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 in 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.

These three models differ primarily in how 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 Command Line Interpreter (CLI) Function 150 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 so that if the process executes a Far Return instruction, it will terminate. The CLI function sets the Instruction Pointer (IP) Register 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.

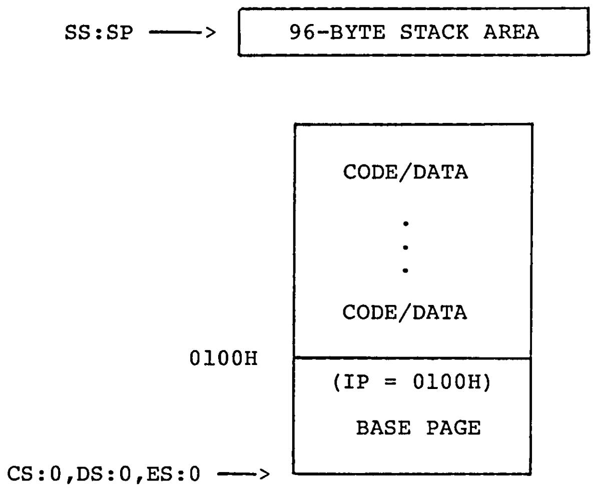


**Figure 4-1. Concurrent CP/M-86 8080 Memory Model**

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 assembly language 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. 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.

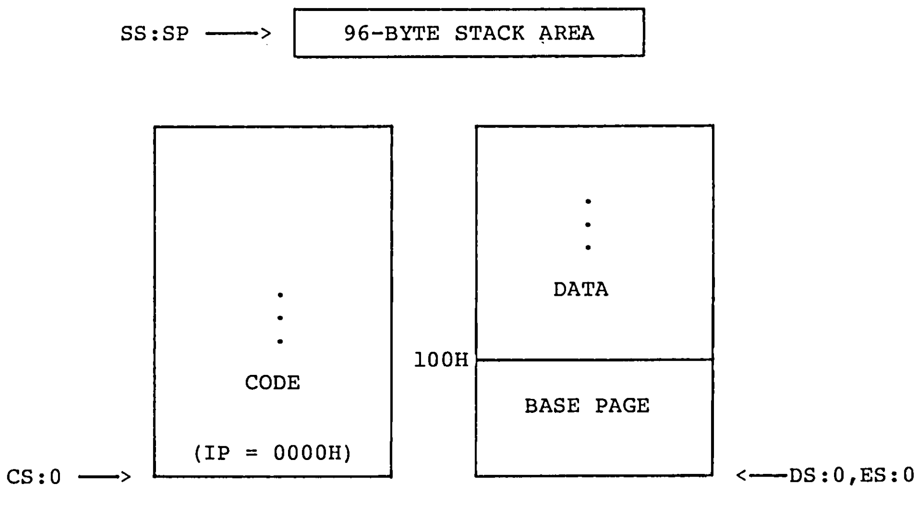


**Figure 4-2. Concurrent CP/M-86 Small Memory Model**

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 assembly language 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, allowing access to the entire memory space.

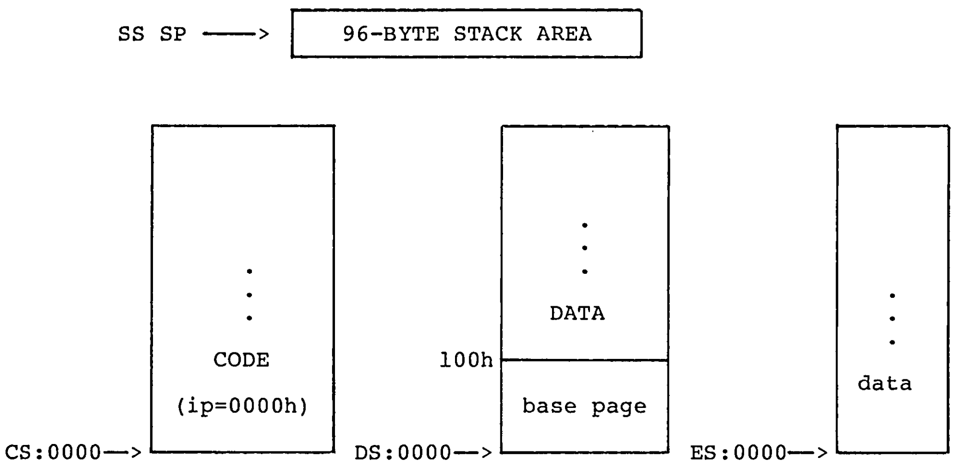


**Figure 4-3. Concurrent CP/M-86 Compact Memory Model**

If the assembly language 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 appears that the SS and SP registers should be set to address the stack group, there are two contradictions. First, the assembly language transient program might 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 assembly language 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 assembly language 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 nondestructive. It does not alter the original HEX file. GENCMD has the following form

```
        GENCMD filename {parameter-list}
```

where the filename corresponds to the HEX input file with an assumed and unspecified filetype 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, allowing intermixed code and data in a single segment. The form of this command is:

```
        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 that defines 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 that must be located at an absolute location. This value is not usually specified, as Concurrent CP/M-86 cannot ensure that the required memory region is available. In that case the CMD file cannot be loaded.

- The B value is used when GENCMD processes a HEX file produced by Intel's OH86 or a 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 usually 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 because 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. The total space required for the group is defined by the range between the lowest and highest data byte addresses. The data group, however, might contain uninitialized storage at the end of the group. 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 might 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 assembly language 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 might be nonzero. 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 OFFF0H 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 its equivalent, byte address 400H. The data group requires a minimum of 300H bytes, but can use up to OFFF0H 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 MCS-86 Software Development Utitities Operating Instructions for ISIS-II Users, published by Intel.) The CMD file produced by GENCMD contains a header record defining 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 |

**Figure 4-4. Intel HEX File Format**

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:<br>00 data record, loaded starting at offset aaaa from current base paragraph<br>01 end of file, cc always = 0FFH<br>02 extended address, aaaa is paragraph base for subsequent data records<br>03 starting code address is aaaa (ignored by GENCMD and Concurrent CP/M-86, IP set according to memory model in use)<br><br>The following are output from ASM-86 only:<br><br>81 same as 00, data belongs to Code Segment<br>82 same as 00, data belongs to Data Segment<br>83 same as 00, data belongs to Stack Segment<br>84 same as 00, data belongs to Extra Segment<br>85 paragraph address for absolute Code Segment<br>86 paragraph address for absolute Data Segment<br>87 paragraph address for absolute Stack Segment<br>88 paragraph address for absolute Extra Segment |

**Table 4-1. (continued)**

| Field | Contents |
|---|---|
| d | Data Byte |
| cc | Check Sum (such that check sum and sum of previous digits (not ASCII codes) |

All characters preceding the colon for each record are ignored. (For additional HEX file format information see MCS-86 Absolute Object File Formats, published by Intel.)

End of Section 4

# Section 5
# System Function Calls

This section describes each Concurrent CP/M-86 system function, including the parameters a process must pass when calling the function, and the values the function returns to the process. You should be familiar with the material in Sections 1 through 4 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. 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, these resources 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 function (Function 143) with the termination code set to 00H.

Under CP/M-86, the System Reset function has a further argument that allows a process not to release its memory. This argument places a piece of code into memory that becomes an interface for later programs. Concurrent CP/M-86 does not include this option. Memory segments are not recovered by the system until all processes that own the memory segment have released it.

See Appendix M for a list of returned error codes.

## FUNCTION 1: VIRTUAL CONSOLE INPUT

Read a character from the default virtual 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 virtual console of the calling process. Before attempting the read, Concurrent CP/M-86 internally calls the Attach console function (Function 146) to verify ownership of the virtual console. If the calling process does not own the virtual 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 virtual console when it begins execution.

Function 1 echoes graphic characters read from the virtual console. This includes the carriage return, line-feed, and backspace characters. It expands tab characters (CTRL-I) in columns of eight characters.

Concurrent CP/M-86 checks for special characters typed on the keyboard. These characters are intercepted in real time by the operating system and perform the following special operations:

| | |
|---|---|
| ● CTRL-S | Suspend Console Output |
| ● CRTL-Q | Activate Console Output |
| ● CTRL-C | Terminate Current Process |

These characters are not returned to a program unless the console is in Raw mode.

Function 1 ignores the terminate character (CTRL-C) if the calling process cannot terminate. (See Function 143.) Function 1 does not return until a character is typed on the virtual console. The system suspends the calling process until a character is ready.

## FUNCTION 2: VIRTUAL CONSOLE OUTPUT

Write a character to the default virtual console

```
Entry Parameters:
    Register  CL: 02H
              DL: ASCII character
```

The Console Output function writes the specified character to the calling process's default virtual console. As in the Virtual Console Input function (Function 1), Concurrent CP/M-86 verifies that the calling process owns its default console before performing the operation. On output, Function 2 expands tabs in columns of eight characters.

## FUNCTION 3: RAW CONSOLE INPUT

Read a character from the default virtual 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 virtual console of the calling process. As in the Virtual Console Input function (Function 1), Concurrent CP/M-86 verifies ownership of the virtual console before performing the operation. Calling Function 3 places the process in Raw mode. No checking is done for special characters such as the terminate character. Characters are not echoed when typed.

**Note:** The process is taken out of Raw mode as soon as a it calls a nonraw virtual console function. Calling Raw Virtual Console Input forces the process to relinquish the CPU resource until a character is typed at the virtual console.

```
FUNCTION 4:  RAW CONSOLE OUTPUT

   Write a character to the default
    virtual console in Raw Mode

Entry Parameters:
    Register  CL:  04H
              DL:  Character
```

The Raw Console Output function writes a character to the default virtual console of the calling process. Concurrent CP/M-86 verifies ownership of the virtual console before permitting the operation. Calling Function 4 places the process in Raw mode. No checking is done for special characters such as the terminate character (CTRL-C).

| FUNCTION 5: LIST OUTPUT |
|---|
| Write a character to the default List device |
| Entry Parameters:<br>Register CL: 05H<br>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 virtual 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 virtual console. Concurrent CP/M-86 verifies that the calling process owns its default virtual 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 | virtual console input status command (If no character if ready, a 00H is returned.) |
| 0FEH | virtual console status command (On return, register AL contains 00H if no character is ready; otherwise it contains 0FFH.) |
| 0FDH | virtual console input command (If no character is ready, the calling process waits until one |

|  |  |
|---|---|
|  | is typed.) Input characters are not echoed to the screen. |
| ASCII character | If the parameter is less than 0FDH, then Function 6 assumes register DL contains a valid ASCII character and sends it to the virtual 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 allow totally transparent I/O when used in conjunction with the virtual 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. Special characters, such as the terminate character, are not intercepted.

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

Concurrent CP/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 (024H). Function 9 writes the string to the calling process's default virtual console. Concurrent CP/M-86 verifies that the calling process owns the virtual console before writing the string. Function 9 expands tabs in columns of eight characters as in the Console Output function (Function 2). The Print String function sets the virtual console to a nonraw state.

Use the Print String function, rather than the single-character functions, whenever possible. The CPU overhead involved in handling the first character is the same as that for a single-character function, but subsequent characters require as little as one-fifth the CPU overhead.

|  |
|---|
| FUNCTION 10: READ CONSOLE BUFFER |
| Read an edited line from the default virtual console |
| Entry Parameters:<br>Register CL: 0AH<br>DX: BUFFER Address - Offset<br>DS: BUFFER Address - Segment |

The Read Console Buffer function reads characters from the calling process's default virtual console and places them into the specified buffer. The format of the buffer is shown in Figure 5-1. Function 10 performs line-editing functions on the line as it is read from the virtual console. The Read Console Buffer function completes a line and returns upon receiving a terminator character from the virtual console or when the maximum number of characters is reached. As in Function 1, the Read Console Buffer function echoes all graphic characters read from the virtual console. Concurrent CP/M-86 verifies that the calling process owns its default virtual console before allowing I/O to begin.

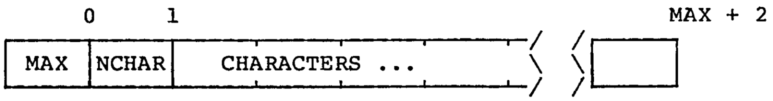


**Figure 5-1. Console Buffer Format**

| | |
|---|---|
| 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 virtual 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.

**Table 5-1. Read Console Buffer Line-editing Characters**

| Character | Function |
|---|---|
| RUB/DEL | Removes the last character from the line and echoes it. |
| (CTRL-E) | Echoes new line, a carriage return (CTRL-M) and a linefeed (CTRL-J), to the screen but does not affect the line buffer. |
| BACKSPACE (CTRL-H) | Removes the last character from the line and backspaces over that character. |
| TAB (CTRL-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 (CTRL-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 (CTRL-M) | Terminates the input line. |
| REDRAW (CTRL-R) | Retypes the current line after echoing a new line. |
| (CTRL-U) | Removes all of the current line from the line buffer, echoes a new line, and starts all over again. |

**Table 5-1. (continued)**

| | |
|---|---|
| (CTRL-X) | Removes all of the current line from the line buffer and echoes enough backspaces to return to the beginning of the line. |
| TERMINATE (CTRL-C) | Asks you if you want to end the running process, if the process can be terminated. Otherwise CTRL-C's are ignored. Function 10 recognizes the terminate character only if it is the first character in the line. |

```
FUNCTION 11:  CONSOLE STATUS

    Obtain the status of the
    default virtual console

Entry Parameters:
    Register  CL: 0BH

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 virtual console of the calling process. If the calling process is not attached to its default virtual console, the Console Status function will cause a dispatch to occur and return 00H (the Not Ready condition).

This function sets the console to the Nonraw mode, allowing recognition of special control characters such as the terminate character, CTRL-C. Use Function 6, Direct Console I/O, to obtain console status in Raw mode.

```
FUNCTION 12:  RETURN BDOS VERSION NUMBER

        Return BDOS Version Number

Entry Parameters:
    Register  CL: 0CH

Return Values:
    Register  AL: 30 (BDOS Version 3.0)
              AH: 14 (Concurrent CP/M-86)
              BX: Same as AX
```

The Return Version Number function returns the BDOS file system version number, allowing version independent programming.

The Return CCPM Version function (Function 163) can be called to obtain the Concurrent CP/M-86 version or revision number. Function 12 indicates the type of operating system but not the revision level.

```
AL = BDOS Version

AH = CPU Type (High Nibble)

       0 = 8080
       1 = 8086

     OS Type  (Low Nibble)

       0 = CP/M              2 = CP/M w/networking
       1 = MP/M              3 = MP/M w/networking
       4 = Concurrent CP/M   6 = Concurrent CP/M
5,7 to E = Reserved                 w/networking
```

**Figure 5-2. Version Number Format**

```
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 Concurrent CP/M-86. If another process has an open file on a removable 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 00H. Otherwise, it returns 0FFH. If the BDOS is not in the Return Error mode when an error occurs (see Function 45), the system displays an error message at the console, identifying the process owning an open file.

```
FUNCTION 14:  SELECT DISK

   Set calling process' 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, continuing through 15 for drive P in a full 16-drive system. Function 14 also 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 00H 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:

| | | |
|---|---|---|
| 01H | : | Permanent Error |
| 04H | : | 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
```

**Note:** See Section 2.4, "File Control Block Definition", for a description of the FCB (in Figure 2-1) and further information concerning it.

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 00H. The calling process passes the address of the FCB, with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the filename and filetype, and byte 12 specifying the extent. The process usually initializes byte 12 to 00H. 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 set the current record field of the FCB (cr) to 00H if the file is to be accessed sequentially from the first record.

See "Compatibility Attributes", Section 2.4.

The Open File function performs the following steps for files opened in Locked or Read-Only mode. If the current user is nonzero 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 00H. 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 00H. 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: Function 15 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. This item remains in the system Lock list until the file is permanently closed or until 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 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 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:

| | | |
|---|---|---|
| 01H | : | Permanent error |
| 04H | : | Select error |
| 05H | : | File is open by another process or by the current process in an incompatible mode |
| 07H | : | File password error |
| 09H | : | ? in the FCB filename or filetype |
| 0AH | : | Process open file limit exceeded |
| 0BH | : | 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 attributes f5' and f6' specify how the file is to be closed, as shown below:

```
f6' = 0, f5' = 0 - Permanent Close (Default mode)
f6' = 0, f5' = 1 - Partial Close
f5' = 0, f6' = 1 - Extended Lock
```

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 if the FCB does not contain new information, the directory update step is bypassed. In this latter case, only read 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'=0 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 lets a process maintain a lock on a file even after the file is closed. The process then can rename, set attributes, or delete a file after the file is closed, without interference from other processes. Setting f6'= 1 before the close operation implements the extended file lock. Resetting f6'= 0 (the default action) clears the file from the lock list unless f5'=1. See Section 2.10.

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 00H to 03H if the close was successful, or 0FFH 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:

| | | |
|---|---|---|
| 01H | : | Permanent Error |
| 02H | : | Read-Only Disk |
| 04H | : | Select Error |
| 06H | : | 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. Byte 12 is usually set to 00H. An ASCII question mark (63, or 03FH hexadecimal) 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 follow either 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 usually 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.

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 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., Read Only at the AL register left 5 bits). Although not 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:

| | |
|---|---|
| 01H | : Permanent Error |
| 04H | : 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 AL, 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 filetype can contain wildcard file specifications (i.e., question marks in bytes f1 through t3), but the dr byte cannot be a wildcard as it can be 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 it must 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 XFCBs. This applies to both types of delete operations.

A process can delete a file that it currently has open if the file was opened in Locked mode. However, the BDOS returns a checksum error if the process makes a subsequent reference to the file with a BDOS function requiring an open FCB. No process can delete files open in R/O or Unlocked mode.

Upon return, the Delete File function returns a directory code in register AL with the value 00H to 03H if the delete was successful or 0FFH 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:

| | | |
|---|---|---|
| 01H | : | Permanent Error |
| 02H | : | Read-Only disk |
| 03H | : | Read-Only file |
| 04H | : | Select Error |
| 05H | : | File open by another process or open in Read-Only or Unlocked mode |
| 07H | : | 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 the current record field in byte cr of the FCB, relative to the current 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 00H for the next read operation. The calling process must set the cr field to 00H 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:

| | | |
|---|---|---|
| 01H | : | Reading unwritten data (end-of-file) |
| 09H | : | Invalid FCB |
| 0AH | : | FCB checksum error |
| 0BH | : | Unlocked file verification error |
| 0FFH | : | Physical error; refer to register AH |

The function returns Error Code 01H if no data exists at the next record position of the file. The no data situation is usually encountered at the end of a file. However, it can also occur if you try 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 09H 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 0AH if the referenced FCB failed the FCB checksum test.

The function returns Error Code 0BH 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 0FFH 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:

| | |
|---|---|
| 01H : | Permanent Error |
| 04H : | 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 set to 0 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. A BDOS Open or Make File function call must have previously activated the referenced FCB.

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 00H 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 00H 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 00H if the write operation was successful. Otherwise, register AL contains an error code identifying the error as shown below:

| | | |
|---|---|---|
| 01H | : | No available directory space |
| 02H | : | No available data block |
| 08H | : | Record locked by another process |
| 09H | : | Invalid FCB |
| 0AH | : | FCB checksum error |
| 0BH | : | Unlocked file verification error |
| 0FFH | : | Physical error; refer to register AH |

The function returns Error Code 01H 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 02H 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 08H if it attempts to write to a record locked by another process. The function returns this error only for files open in Unlocked mode.

The function returns Error Code 09H 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 0AH if the referenced FCB failed the FCB checksum test.

The function returns Error Code 0BH if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. The function returns this error only for files open in Unlocked mode.

The function returns Error Code 0FFH 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:

```
01H : Permanent Error
02H : Read-Only disk
03H : Read-Only file or
      File open in Read-Only mode or
      File password protected in Write mode
04H : 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 set to 0 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 starts 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 filename and filetype, and byte 12 set to the extent number. Byte 12 is usually set to 00H. Byte 32 of the FCB (the cr field) must be initialized to 00H (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 - Do not 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 the list below, from Function 102).

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 |

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 file 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 initializes all file attributes to 0.

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 automatically creates 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 00H through 03H if the make operation was successful, or 0FFH if no directory space was available. Register AH is set to 00H in both 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:

| | |
|---|---|
| 01H : | Permanent Error |
| 02H : | Read-Only disk |
| 04H : | Select Error |
| 08H : | File already exists |
| 09H : | ? in file name or type field |
| 0AH : | Process open file limit exceeded |
| 0BH : | 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 it must 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 specific, 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 00H to 03H if the rename was successful, or 0FFH if the file named by the first filename in the FCB was not found. Register AH is set to 00H 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 with register AH set to one of the following physical or extended error codes:

```
01H : Permanent Error
02H : Read-Only disk
03H : Read-Only file
04H : Select Error
05H : File open by another process
07H : File password error
08H : File already exists
09H : ? in filename or filetype
```

## 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 either by an explicit BDOS Select Disk call (Function 14), or by an implicit selection when a BDOS file operation specifies a non-00H 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: Disk Number
              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
```

Direct Memory Address (DMA) often refers to disk controllers that directly access the memory of the computer to transfer data to and from the disk subsystem. Under Concurrent CP/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
```

Concurrent CP/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 can 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.

You can use Get Disk Free Space (Function 46) to directly return the number of free 128-byte records on a drive. In fact, the Concurrent CP/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. You must perform a successful drive reset operation to restore a Read-Only drive to the Read-Write state. (See Functions 13 and 37.)

The Write Protect Disk function is conditional under Concurrent CP/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 00H. 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 Concurrent CP/M-86 that detect changed disk media.

The format of the bit vector is analogous to that of the log-in vector returned by Function 24. The least significant bit corresponds to drive A; 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', the compatibility attributes, 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 not use a wildcard file specification. Also, if the specified file is password protected, the correct password must be placed in the first eight bytes of the current DMA buffer or it must 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 filename and filetype. The function then updates the directory to contain the selected indicators. File attributes t1', t2', and t3' are reserved as compatibility attributes. Attributes f1' through f4' are described in Section 2.4.1. Indicators f5' through f8' are reserved for use as interface attributes.

This function is not performed if the referenced FCB specifies a file 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 values 00H to 03H if the function was successful, or 0FFH if the file specified by the referenced FCB was not found. Register AH is set to 00H in both 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:

| | | |
|---|---|---|
| 01H | : | Permanent Error |
| 02H | : | Read-Only disk |
| 04H | : | Select error |
| 05H | : | File open by another process |
| 07H | : | File password error |
| 09H | : | ? in filename or filetype |

```
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. 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 Error modes (See Function 45), Function 31 returns the value 0FFFFH.

```
FUNCTION 32:  SET/GET USER CODE

          Set or 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 GET
              BL: Same as AL
```

A process can change or interrogate its current default user number by calling Function 32. If register DL = 0FFH, then the function returns the value of this user number in register AL. The value can range from 0 to 0FH. If register DL is not 0FFH, then the function changes the default user number to the value of DL (modulo 10H).

Under Concurrent CP/M-86, a new process inherits its initial default user code from its parent, the process creating the new process. Changing the default user code does not change the user code of the parent. On the other hand, all child processes of the calling process inherit the new user code.

This convention is demonstrated by the operation of the TMP. When a command is typed, a new process is created with the same user code as that of the TMP. If this new process changes its user code, the TMP is unaffected. Once the new process terminates, the TMP exhibits the same user code in its prompt as before the command was entered and the child process created.

## 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 position 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 262,143. 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 mighty or might not contain data). Function 33 places the specified record number in the random record field. 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 rereads the same record. After a random read operation, a file can be accessed sequentially, starting from the current randomly accessed position. However, the last randomly accessed record is reread or rewritten 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 00H if the read operation was successful. Otherwise, register AL contains one of the following error codes:

|  |  |  |
|---|---|---|
| 01H | : | Reading unwritten data |
| 03H | : | Cannot close current extent |
| 04H | : | Seek to unwritten extent |
| 06H | : | Random record number out of range |
| 0AH | : | FCB checksum error |
| 0BH | : | Unlocked file verification error |
| 0FFH | : | Physical error: refer to register AH |

The function returns Error Code 01H when it accesses a data block not previously written. This may indicate an end-of-file (EOF) condition.

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

The function returns Error Code 04H when a read random operation accesses an extent not previously created.

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

The function returns Error Code 0AH if the referenced FCB failed the FCB checksum test.

The function returns Error Code 0BH 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 0FFA 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:

|  |  |  |
|---|---|---|
| 01H | : | Permanent Error |
| 04H | : | 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 set to 0 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 analogous to the Read Random Function, except that data is written to the disk from the current DMA address. If the disk extent and/or data block where the data is to be written is not already allocated, the BDOS automatically performs the allocation before the write operation continues.

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 might or might not contain 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 reread or rewritten 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 00H if the write operation was successful.

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

| | |
|---|---|
| 02H | : No available data block |
| 03H | : Cannot close current extent |
| 05H | : No available directory space |
| 06H | : Random record number out of range |
| 08H | : Record locked by another process |
| 0AH | : FCB checksum error |
| 0BH | : Unlocked file verification error |
| 0FFH | : Physical error: refer to register AH |

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

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

The function returns Error Code 05H 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 06H when byte 35 (r2) of the referenced FCB is greater than 3.

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

The function returns Error Code 0AH if the referenced FCB failed the FCB checksum test.

The function returns Error Code 0BH if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. The function returns this error only for files open in Unlocked mode.

The function returns Error Code 0FFH 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:

| | |
|---|---|
| 01H : | Permanent Error |
| 02H : | Read-Only disk |
| 03H : | Read-Only file |
| | File open in Read-Only mode |
| | File password protected in Write mode |
| 04H : | 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 set to 0 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. This is the address of the record immediately following the end of the file. The virtual size of a file corresponds to the physical size if the file is written sequentially. If the file is written in random mode, gaps might exist in the allocation, and the file might contain fewer records than the indicated size. For example, if a single record with record number 262,143 (the Concurrent CP/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 one data block is actually allocated.

To compute file size, the calling process passes the address of an FCB in random mode format (bytes r0, r1, and r2 present). Note that the FCB must contain a specific filename and filetype. 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 04H, then the file contains the maximum record count 262,144.

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

**Note:** the file need not be open in order to use Function 35.

Upon return, Function 35 returns a 00H 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 00H in both 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:

| | | |
|---|---|---|
| 01H | : | Permanent Error |
| 04H | : | Select Error |
| 09H | : | ? in filename or filetype |

```
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 initially to 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, because 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.

Function 36 can also be used 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 to set 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 Concurrent CP/M-86. If another process has a file open on a drive to be reset, and if the drive is removable 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 00H. Otherwise, it sets register AL to 0FFH. If the BDOS is not in Return Error mode (see Function 45), 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. 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 decimal values.

| | |
|---|---|
| 0AH | Process Open File limit exceeded |
| 0BH | No room in the system Lock list |

If the Access Drive function is successful, it sets register AL to 00H.

## 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 can be lost.

## FUNCTION 40: WRITE RANDOM WITH ZERO FILL

Write a Random Record to a disk File
and Prefill 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 00H Fill function is similar to the Write Random function (Function 34) with the exception that it fills a previously unallocated data block with nulls 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 useful when more than one process has Read-Write access to a common file. This situation is supported under Concurrent CP/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 Concurrent CP/M-86 and CP/M-86.

Upon return, the Test And Write Record function sets register AL to 00H if the function was successful.

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

```
          01H : Reading unwritten data
          03H : Cannot close current extent
          04H : Seek to unwritten extent
          06H : Random record number out of range
          07H : Records did not match
          08H : Record locked by another process
          0AH : FCB checksum error
          0BH : Unlocked file verification error
          0FFH : Physical error : refer to register AH
```

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

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

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

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

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

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

The function returns Error Code 0AH if the referenced FCB failed the FCB checksum test.

The function returns Error Code 0BH if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. The function returns this error only for files opened in Unlocked mode.

The function returns Error Code 0FFH 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:

```
01H : Permanent Error
02H : Read-Only disk
03H : Read-Only file or
      File open in Read-Only mode
      File password protected in Write mode
04H : 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 set to 0 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 provides compatibility between Concurrent CP/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.

Each locked record consumes an entry in the BDOS system Lock list that is shared by locked record and open file entries. If there is not sufficient space in the system Lock list to lock all the specified records, or if 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 00H if the lock operation was successful. Otherwise, register AL contains one of the following error codes:

| | |
|---|---|
| 01H | : Reading unwritten data |
| 03H | : Cannot close current extent |
| 04H | : Seek to unwritten extent |
| 06H | : Random Record Number out of range |
| 08H | : Record locked by another process |
| 0AH | : FCB checksum error |
| 0BH | : Unlocked file verification error |
| 0CH | : Process record lock limit exceeded |
| 0DH | : Invalid File ID |
| 0EH | : No room in the system Lock List |
| 0FFH | : Physical error: refer to register AH |

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

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

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

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

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

The function returns Error Code 0AH if the referenced FCB failed the FCB checksum test.

The function returns Error Code 0BH 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 0CH 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.

The function returns Error Code 0DH when an invalid File ID is placed in the current DMA.

The function returns Error Code 0EH when the system Lock list is full or when the calling process has exceeded the maximum lock records per process implimented in this installation.

The function returns Error Code 0FFH if a physical error was encountered and the BDOS Error mode is either 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:

```
01H : Permanent Error
04H : 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 set to 0 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, skipping those records locked by other processes.

Upon return, the Unlock Record function sets register AL to 00H if the unlock operation was successful.

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

| | |
|---|---|
| 01H : | Reading unwritten data |
| 03H : | Cannot close current extent |
| 04H : | Seek to unwritten extent |
| 06H : | Random Record Number out of range |
| 0AH : | FCB checksum error |
| 0BH : | Unlocked file verification error |
| 0CH : | Invalid File ID |
| 0FFH : | Physical error: refer to register AH |

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

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

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

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

The function returns Error Code 0AH if the referenced FCB failed the FCB checksum test.

The function returns Error Code 0BH 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 0CH when an invalid File ID is placed in the current DMA.

The function returns Error Code 0FFH 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:

| | |
|---|---|
| 01H : | Permanent Error |
| 04H : | 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 set to 0 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 Concurrent CP/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 00H 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 (Section 2.15) 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 Error modes, BDOS sets register AL to 0FFH, 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, the Error mode is set to Return Error mode. If register DL is set to 0FEH, 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, continuing 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 5-3.

|  |  |  |
|---|---|---|
| fs0 | fs1 | fs2 |

**Figure 5-3. Disk Free Space Field Format**

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

Upon return, the function sets register AL to 00H 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 AH to one of the following values:

```
01H - Permanent Error
04H - 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 Concurrent CP/M-86, the Chain To Program function releases the memory of the calling process 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, Concurrent CP/M-86 loads the new program into the same memory area as the old program. 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.

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 Function 150.

The Chain To Program function returns an error if no CMD file is found. If a CMD file is found and an error occurs after it is successfully opened, the calling process terminates, as its memory has been released.

## FUNCTION 48: FLUSH BUFFERS

Flush 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.

Upon return, the function sets register AL to 00H 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 01H : 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 5-4. BIOS Descriptor Format**

The Direct BIOS Call function is provided under Concurrent CP/M-86 for compatibility with programs generated under CP/M-86 that use this function. Under Concurrent CP/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 Concurrent CP/M-86 XIOS.

**Note:** Calls to the XIOS Console Status, Input, and Output functions do not go to the XIOS if the referenced device is a virtual console.

## 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 5-5. 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. |

In CP/M-86, Functions 53 and 54 do not allocate memory. Under Concurrent CP/M-86, these functions allocate memory when called because other processes are competing for common memory. For compatibility with CP/M-86, the ALLOC ABS MEM function (Function 56) does not return an error if there is a memory segment allocated at the absolute address.

Function 53 allocates the largest available memory region that 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.

See Appendix M for a list of error codes returned in CX.

**Note:** functions 53, 54, 55, 56, 57, and 58 are included for compatibility with CP/M-86. If you're developing software for Concurrent CP/M-86, use Functions 128 and 129.

```
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 5-6. 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 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. |

In CP/M-86, Functions 53 and 54 do not allocate memory. Under Concurrent CP/M-86, these functions allocate memory when called, because other processes are competing for common memory. For compatibility with CP/M-86, the ALLOC ABS MEM function (Function 56) does not return an error if there is a memory segment allocated at the absolute address.

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.

See Appendix M for a list of error codes returned in CX.

**Note:** functions 53, 54, 55, 56, 57, and 58 are included for compatibility with CP/M-86. If you're developing software for Concurrent CP/M-86, use Functions 128 and 129.

## 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
                  0FFH on failure
              BL: Same as AL
              CX: Error Code
              MCB filled in
```

Memory Control Block (MCB):

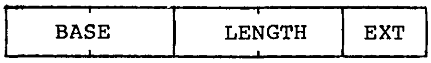


**Figure 5-7. Memory Control Block Format**

| | |
|---|---|
| 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.

See Appendix M for a list of error codes returned in CX.

**Note:** functions 53, 54, 55, 56, 57, and 58 are included for compatibility with CP/M-86. If you are developing software for Concurrent CP/M-86, use Functions 128 and 129.

## FUNCTION 56: ALLOC ABS MEM

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):

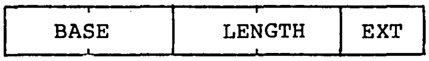


**Figure 5-8. Memory Control Block Format**

| | |
|---|---|
| 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 that starts at the address specified by the BASE field. The memory area's 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. If the calling process already owns the requested memory, no error is returned. This assures compatibility with CP/M-86.

See Appendix M for a list of error codes returned in CX.

**Note:** functions 53, 54, 55, 56, 57, and 58 are included for compatibility with CP/M-86. If you are developing software for Concurrent CP/M-86, use Functions 128 and 129.

```
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 5-9. 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 00H, the function releases the memory area beginning at the specified BASE and ending at the end of the previously allocated memory segment.

See Appendix M for a list of error codes returned in CX.

**Note:** functions 53, 54, 55, 56, 57, and 58 are included for compatibility with CP/M-86. If you are developing software for Concurrent CP/M-86, use Functions 128 and 129.

| FUNCTION 58: FREE ALL MEM |
|---|
| Free All Memory Owned By the Calling Process |
| Entry Parameters:<br>Register CL: 3AH |

In the Concurrent CP/M-86 environment, the Free All Memory function releases all of the calling process' memory except the User Data Area (UDA). This function is useful for system processes and for subprocesses that share the memory of another process.

This function should NOT be used by processes running programs loaded into the Transient Program Areas.

**Note:** functions 53, 54, 55, 56, 57, and 58 are included for compatibility with CP/M-86. If you are developing software for Concurrent CP/M-86, use Functions 128 and 129.

```
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
                  0FFFFH on error
              BX: Same as AX
              CX: Error Code
```

The Program Load function loads disk file of type CMD into memory. Upon entry, register DX contains the DS-relative offset of a successfully opened FCB that specifies the input CMD file. Upon return, register AX has the value 0FFFFH 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 program, and the DMA offset address to 0080H. Note that 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.

**Note:** open the .CMD file in Read-Only mode and close it once the load is completed.

See Appendix M for a list of error codes returned in CX.

## 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 directory label. The extent field of the FCB (byte 12) contains the user's specification of the directory label data byte. The definition of the directory 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

      (Bit 0 is the least significant bit)
```

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 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 if no space existed in the referenced directory to create a directory label. Register AH is set to 00H 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:

| | | |
|---|---|---|
| 01H | : | Permanent Error |
| 02H | : | Read-Only disk |
| 04H | : | Select Error |
| 07H | : | 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 DL with 0 for drive A, 1 for drive B, continuing 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

        (Bit 0 is the least significant bit)
```

Function 101 returns the directory label data byte to the calling process in register AL. Register AL equal 00H 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 Error modes (see Function 45), it returns with register AL set to 0FFH and register AH set to one of the following:

```
01H : Permanent Error
04H : 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 00H to 03H if the XFCB read operation was successful, or 0FFH if the XFCB was not found. Register AH is set to 00H 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:

| | |
|---|---|
| 01H : | Permanent Error |
| 04H : | 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 00H to 03H if the XFCB create or update was successful, or 0FFH if no directory label existed on the specified drive, or the file specified in the FCB was not found, or no space existed in the directory to create an XFCB. Register AH is set to 00H 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:

| | |
|---|---|
| 01H : | Permanent Error |
| 02H : | Read-Only disk |
| 04H : | Select Error |
| 07H : | 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 Concurrent CP/M-86, this function also sets the second field of the system date and time to 00H.

```
FUNCTION 105:  GET DATE AND TIME

              Get 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 that 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 Concurrent CP/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. Concurrent CP/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 Concurrent CP/M-86 serial number to the addressed, six-byte SERIAL field as a six-byte ASCII numeral.

```
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 | * 0000H | * 0000H |
|---|---|---|---|---|

**Figure 5-10. Memory Parameter Block (MPB)**

| | |
|---|---|
| START | if non-00H, an absolute request at this paragraph |
| MIN | minimum memory needed (paragraphs) |
| MAX | maximum memory wanted (paragraphs) |
| * 0000H | these fields must be 00H; they are used internally. |

The Memory Allocation function allows a program to allocate extra memory. A successful allocation allocates 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; Concurrent CP/M-86 also releases all memory owned by a process at termination.

**Note:** MIN and MAX fields must be explicitly filled in. The MAX value must be greater than or equal to the MIN value.

See Appendix M for a list of error codes returned in CX.

## 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 | * 0000H |

**Figure 5-11. 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.

The * 0000H field must be initialized to zero.

Under certain circumstances, Concurrent CP/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.

See Appendix M for a list of error codes returned in CX.

```
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 noninterrupt driven devices. It should be used whenever the XIOS is waiting for a noninterrupt event. The calling process relinquishes the CPU and allows Concurrent CP/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, and they return whether the device is ready or not. When the device is ready, Concurrent CP/M-86 restores the calling process to the RUN state and returns. Upon return, the calling process knows the device is ready.

See Appendix M for a list of error codes returned in CX.

```
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 occurred, or an error has occurred. 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 Concurrent CP/M-86 reserves flags 0, 1, 2, and 3 for system use.

See Appendix M for a list of error codes returned in CX.

```
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 occurred. A process waiting for this flag is placed back into the RUN state. If there are no processes waiting, then the next process to wait for this flag returns 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.

See Appendix M for a list of error codes returned in CX.

## 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 5-12. Queue Descriptor (QD) Format**

| | |
|---|---|
| FLAGS | Queue Flags. The bits are defined as follows:<br><br>0001H - Mutual exclusion queue<br>0002H - CANNOT be deleted<br>0004H - restricted to SYSTEM processes<br>0008H - RSP message queue<br>0010H - Used internally<br>0020H - RPL address queue<br>0040H - Used internally<br>0080H - Used internally<br><br>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 <u>must</u> 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. |
| * 0000H | for internal use. Must be initialized to zero. |

Every system queue under Concurrent CP/M-86 is associated with a Queue Descriptor that resides within the Concurrent CP/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 Concurrent CP/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.

See Appendix M for a list of error codes returned in CX.

The buffer for a system queue must also reside within the System Data area. For non-00H 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.

All system queues must have unique names. The function returns an error code if a system queue already exists by the given name.

Under Concurrent CP/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
```

| | | | |
|---|---|---|---|
| * 0000H | QUEUEID | * 0000H | BUFFER |
| NAME | | | |

**Figure 5-13. Queue Parameter Block (QPB)**

| | |
|---|---|
| QUEUEID | modified by Open Queue |
| * 0000H | Reserved for internal use; must be initialized to zero |
| BUFFER | not used for Open Queue |
| NAME | 8-byte system queue name. |

All system queues under Concurrent CP/M-86 must be explicitly 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. See Appendix M for a list of error codes returned in CX.

```
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
```

| | | | |
|---|---|---|---|
| * 0000H | QUEUEID | * 0000H | BUFFER |
| NAME | | | |

**Figure 5-14. Queue Parameter Block (QPB)**

| | |
|---|---|
| QUEUEID | filled in by a previous Open Queue |
| * 0000H | Reserved for internal use; must be initialized to zero |
| 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 has not been previously opened.

See Appendix M for a list of error codes returned in CX.

## 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
```

| | | | |
|---|---|---|---|
| * 0000H | QUEUEID | * 0000H | BUFFER |
| NAME | | | |

**Figure 5-15. Queue Parameter Block (QPB)**

| | |
|---|---|
| QUEUEID | filled in by previous Open Queue |
| * 0000H | reserved for internal use; must be initialized to zero |
| 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 there are not enough messages to read from the queue, the calling process waits until another process writes into the queue before returning. See Appendix M for a list of error codes returned in CX.

## 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
```

| | | | |
|---|---|---|---|
| * 0000H | QUEUEID | * 0000H | BUFFER |
| NAME | | | |

**Figure 5-16. Queue Parameter Block (QPB)**

| | |
|---|---|
| QUEUEID | filled in by previous Open Queue |
| * 0000H | Reserved for internal use; must be initialized to zero |
| 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, but it returns an error code if there are not enough messages to read instead of waiting for another process to write to the queue.

See Appendix M for a list of error codes returned in CX.

```
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
```

| * 0000H | QUEUEID | * 0000H | BUFFER |
|---|---|---|---|
| NAME | | | |

**Figure 5-17. Queue Parameter Block (QPB)**

| | |
|---|---|
| QUEUEID | filled in by previous Open Queue |
| * 0000H | reserved for internal use; must be initialized to zero |
| 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 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. See Appendix M for a list of error codes returned in CX.

## 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
```

| | | | |
|---|---|---|---|
| * 0000H | QUEUEID | * 0000H | BUFFER |
| NAME | | | |

**Figure 5-18. Queue Parameter Block (QPB)**

| | |
|---|---|
| QUEUEID | filled in by previous Open Queue |
| * 0000H | reserved for internal use; must be initialized to zero |
| 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 is analagous to the Write Queue function, but it returns an error code if there is not enough System Queue Buffer for the message to be written instead of waiting for another process to read from the queue.

See Appendix M for a list of error codes returned in CX.

| FUNCTION 141: DELAY |
|---|
| Delay for specified number of System Ticks |
| Entry Parameters:<br>Register CL: 08DH<br>DX: Number of System Ticks |

The Delay function causes the calling process to wait a until the specified number of system ticks has occurred. 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 Function 154).

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 might 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:<br>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 Concurrent CP/M-86 Dispatcher is priority driven, with round-robin scheduling of equivalent-priority processes. When a process calls the Dispatch function, it is rescheduled, so 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 cannot 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. Because 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 Prpcess 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 User Data Area (UDA) can be anywhere in memory but is required to be on a paragraph boundary. 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 Concurrent CP/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 | | MEM | |
| 18 | RESERVED | | | | | | PARENT | |
| 20 | CNS | RESERVED | | | LIST | RESERVED | | |
| 28 | RESERVED | | | | | | | |

**Figure 5-19. Process Descriptor (PD) Format**

| | |
|---|---|
| LINK | link field for insertion on current system list. If this field's initial value is nonzero, 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 is always 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 Concurrent CP/M-86 systems. User processes priorities should be from 200-254.

| | |
|---|---|
| 1 | Initialization Process |
| 2 - 31 | Interrupt Handlers |
| 32 - 63 | System Processes |
| 64 - 189 | Undefined |
| 191 - 197 | Undefined |
| 198 | Terminal Message Process |
| 199 | Undefined |
| 200 | Default Priority For Transients |
| 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 Concurrent CP/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 is copied into the PD from the PD table. When this process terminates, the PD is 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 depending on console calls. |
| 080H | |

| | | |
|---|---|---|
| | CTRL-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. Used internally. |

| | |
|---|---|
| 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 |
| MEM | Root of linked list of Memory Segment Descriptors that are owned by this process. Initialized to zero. |
| PARENT | Process that created this process. The Create Process function sets this value at process creation. The parent field is set to zero if the parent terminates before the child. |
| 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). |

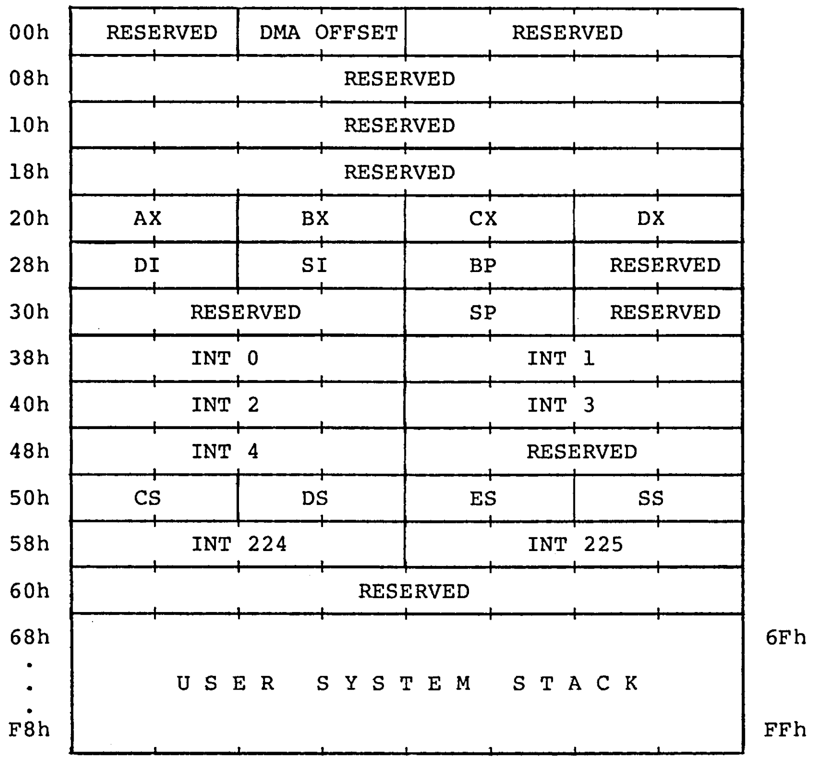


**Figure 5-20. User Data Area (UDA)**

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 is filled in with the initial Code Segment (see CS Below) and the initial flags. The initial flags are set to 0200H, which means that interrupts are on, and all other flags are off. Concurrent CP/M-86 starts a new process by executing an Interrupt Return instruction with the initial stack.

**Note:** this stack area is distinct from the User System Stack at the end of the UDA.

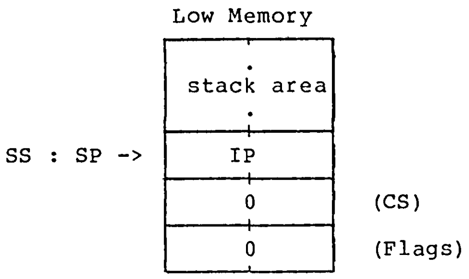


**Figure 5-21. Stack Initialization Area**

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 that services these interrupts. Those fields that are zero are 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 process' Data Segment.

INT 224,
INT 225 Interrupts 224 and 225 are used to communicate with Concurrent CP/M-86 by typical programs. These interrupt vectors are 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 Concurrent CP/M-86 calls that its children make. The suggested protocol is to keep INT 225 pointing to the

Concurrent CP/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 Concurrent CP/M-86 using INT 225 for the actual Concurrent CP/M-86 call.

| | |
|---|---|
| RESERVED | These fields are used internally and must be intialized to zero. |
| USER SYSTEM STACK | This is the stack area used by the process when it is in the operating system. The SP variable in the UDA should not point to this area. |

See Appendix M for a list of error codes returned in CX.

```
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 afterwards can run at a lower priority.

The best or highest priority is 00H while the worst or lowest priority is 0FFH. Transient processes are initialized to run at C8H (200 decimal) by the Send CLI function.

See Appendix M for a list of error codes returned in CX.

## FUNCTION 146: ATTACH CONSOLE

Attach default virtual console
to calling process

```
Entry Parameters:
    Register  CL: 092H

Return Values:
              CX: Error Code
```

The Attach Console function attaches the default virtual console to the calling process. If the virtual console is already owned by the calling process or if it is not owned by another process, the Attach Console function immediately returns with ownership established and verified. If another process owns the virtual console, the calling process waits until the virtual console becomes available.

When the virtual console becomes free through a Detach Console call, the process that is waiting for the virtual console with the highest priority obtains it. The Attach Console function is called internally by all console I/O functions except the Raw Console functions.

See Appendix M for a list of error codes returned in CX.

```
FUNCTION 147:  DETACH CONSOLE

      Detach default virtual console
          from calling process

Entry Parameters:
    Register  CL: 093H

Return Values:
              CX: Error Code
```

The Detach Console function detaches the default virtual console from the calling process. If the default virtual console is not attached to the calling process, no action is taken. If other processes are waiting to attach to the virtual console, the process with the highest priority attaches the virtual console. If there is more than one process with the same priority waiting for the virtual console, it is given on a first-come first-serve basis.

See Appendix M for a list of error codes returned in CX.

```
FUNCTION 148:  SET CONSOLE

        Set the calling process's
         default virtual 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' default virtual console to the value specified. If the virtual console number specified is not one supported by this particular implementation of Concurrent CP/M-86, the function returns an error code, and does not change the default virtual console. If the virtual console number is valid, the function detaches the previous default virtual console from the calling process. The Set Console function then attaches the new virtual console to the calling process through the Attach Console function. If another process already owns the new virtual console, the calling process waits until the virtual console becomes available.

See Appendix M for a list of error codes returned in CX.

FUNCTION 149: ASSIGN CONSOLE

Assign default virtual console
to another process

```
Entry Parameters:
    Register  CL: 095H
              DX: ACB Address - Offset
              DS: ACB Address - Segment

Return Values:
              CX: Error Code
```

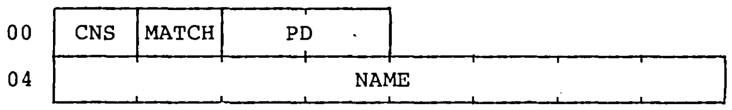


**Figure 5-22. Assign Control Block (ACB)**

| | |
|---|---|
| CNS | Virtual console to assign |
| MATCH | Boolean; if 0FFH, the process being assigned the virtual 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 virtual console. If this field is zero, a search is made of the Thread list for a process whose name is NAME. This field must be either 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 virtual 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 virtual console. The function ignores other processes waiting to attach to the specified virtual console, and they continue to wait until the current owner either calls the Detach function or terminates.

See Appendix M for a list of error codes returned in CX.

```
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
```

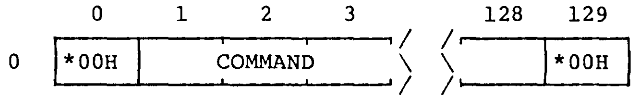


**Figure 5-23. CLI Command Line Buffer**

| | |
|---|---|
| *00H | Must be set to zero for internal use. |
| COMMAND | 1-128 ASCII characters terminated with a null character. |

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 virtual console, the CLI function assigns the virtual console to either the newly created process, or to the Resident System Process (RSP) that acts on the command. The calling process must reattach to its default virtual 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 tries 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' default virtual 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 filetype 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. This search order is discussed in section 2.4 of the Concurrent CP/M-86 Operating System User's Guide. 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 virtual console to the new process and then returns.

The calling process should set its priority to less than the TMP (198) if it wants to attach to the virtual console after the created process releases it. Once the calling process has successfully reattached, it should set its priority back to 200.

See Appendix M for a list of error codes returned in CX.

```
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
```

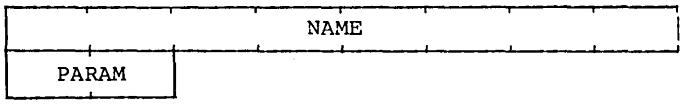


**Figure 5-24. Call Parameter Block (CPB)**

| | |
|---|---|
| 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).

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 reusable, the procedure must write the message just before returning.

See Appendix M for a list of error codes returned in CX.

## 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 5-25. 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 filename. All letters are converted to upper-case. If the name is not eight characters long, the remaining bytes in the filename field are padded with blanks. If the filename has an 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 filetype. If no |

| | |
|---|---|
| | type is specified, the type field is initialized to blanks. All letters are converted to upper-case. If the type is not three characters long, the remaining bytes in the filetype 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 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 looks at the next character. If the character is a null (0AH) 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 nonblank or nontab 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. See Appendix M for a list of error codes returned in CX.

```
FUNCTION 153:  GET CONSOLE

     Return the Calling Process'
       Default Virtual Console

Entry Parameters:
    Register  CL: 099H

Return Values:
              AL: Console number
              BL: Same as AL
```

The Get Console function returns the calling process' default virtual console number.

```
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 Concurrent CP/M-86 uses. See Figure 5-26, System Data Area, below.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00H | RESERVED | | | | | | | |
| 08H | RESERVED | | | | | | | |
| 10H | RESERVED | | | | | | | |
| 18H | RESERVED | | | | | | | |
| 20H | RESERVED | | | | | | | |
| 28H | XIOS ENTRY | | | | XIOS INIT | | | |
| 30H | RESERVED | | | | | | | |
| 38H | DISPATCHER | | | | PDISP | | | |
| 40H | MPMSEG | | RESERVED | | ENDSEG | | RESERVED | NCNS |
| 48H | NLST | RESERVED | N_FLAGS | SYS_DISK | MMP | | RESERVED | DAY FILE |
| 50H | TEMP DISK | TICKS /SEC | RESERVED | | | | | |
| 58H | RESERVED | | MFL | | PUL | | QUL | |
| 60H | RESERVED | | | | | | | |
| 68H | RLR | | DLR | | DRL | | PLR | |
| 70H | RESERVED | | THRDRT | | QLR | | MAL | |
| 78H | VERSION | | VERNUM | | MPMVERNUM | | TOD_DAY | |
| 80H | TOD_HR | TOD_MIN | TOD_SEC | NCON DEV | NLST DEV | RESERVED | RESERVED | |

**Figure 5-26. System Data Area**

| | |
|---|---|
| XIOS ENTRY | Double-word address of the Extended I/O System entry point for intermodule communication. All XIOS function calls go through this entry point. |
| XIOS INIT | Double-word address of the Extended I/O System Initialization entry point. System hardware initialization takes place as calls go through this entry point. |
| DISPATCHER | Double-word address of the Dispatcher entry point that handles interrupt returns. Executing a Far Jump to |

| | |
|---|---|
| | this address is equivalent to executing an Interrupt Return instruction. The Dispatcher routine causes a dispatch to occur and then executes an Interrupt Return. All registers are preserved and one level of stack is used. This function should be used as a exit point by all interrupt routines that use the Flag Set function. |
| PDISP | Double-word address of the Dispatcher entry point that causes a dispatch to occur with all registers preserved. Once the dispatch is done, a RETF instruction is executed. Executing a JMPF PDISP is equivalent to executing a RETF instruction. This function should be executed whenever a resource is released that might be wanted by a waiting process. |
| MPMSEG | Starting paragraph of the operating system area. This is also the Code Segment of the Supervisor Module. |
| ENDSEG | First paragraph beyond the end of the operating system area. |
| NCNS | Number of system consoles. |
| NLST | Number of list devices. |
| NFLAGS | Number of system flags. |
| SYSDISK | Default system diskette. The CLI looks on this diskette if it cannot open the command file on the user's current default diskette. |
| MMP | Maximum memory allowed per process. |
| DAY FILE | Day File option. If this value is 0FFH, log information is displayed on system consoles at each command. |
| TEMP DISK | Default temporary diskette. Programs that create temporary files should use this diskette. |
| TICKS/SEC | The number of system ticks per second. |
| MFL | Link list root of free memory partitions. |
| PUL | Link list root of unused Process Descriptors. |
| QUL | Link list root of unused Queue Descriptors. |
| RLR | Ready List Root. Linked list of PDs that are ready to run. |
| DLR | Delay List Root. Link list of PDs that are delaying for a specified number of system ticks. |

| | |
|---|---|
| DRL | Dispatcher Ready List. Temporary holding place for PDs that have just been made ready to run. |
| PLR | Poll List Root. Linked list of PDs that are polling on devices. |
| THRDRT | Thread List Root. Linked list of all current PDs on the system. The list is threaded though the THREAD field of the PD instead of the LINK field. |
| QLR | Queue List Root. Linked list of all System QDs. |
| MAL | Link list of active memory allocation units. A MAU is created from one or more memory partitions. |
| VERSION | Address relative to MPMSEG of version string. |
| VERNUM | MP/M-86 version number (Function 12). |
| MPMVERNUM | MP/M-86 version number (Function 163). |
| TOD_DAY | Time of Day. Number of days since 1 Jan, 1978. |
| TOD_HR | Time of Day. Hour of the day. |
| TOD_MIN | Time of Day. Minute of the hour. |
| TOD_SEC | Time of Day. Second of the minute. |
| NCONDEV | Number of XIOS consoles. |
| NLSTDEV | Number of XIOS list devices. |
| RESERVED | Reserved for internal use. |

## 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
```

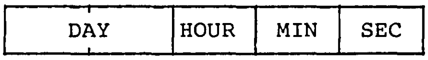


**Figure 5-27. Time Of Day Structure (TOD)**

| | |
|---|---|
| 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'
              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: 0 if successful
                  0FFH on failure
              BL: Same as AL
              CX: Error Code
```

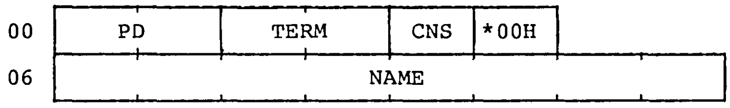


**Figure 5-28. Abort Parameter Block (APB)**

| | |
|---|---|
| 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. |
| *00H | 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 00H 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.

See Appendix M for a list of error codes returned in CX.

```
FUNCTION 158:  ATTACH LIST

     Attach to the Calling Process'
          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.

See Appendix M for a list of error codes returned in CX.

## FUNCTION 159: DETACH LIST

Detach the Calling Process'
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.

See Appendix M for a list of error codes returned in CX.

## FUNCTION 160: SET LIST

Set the Calling Process' 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.

See Appendix M for a list of error codes returned in CX.

## 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 attaches 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 00H 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.

See Appendix M for a list of error codes returned in CX.

## FUNCTION 162: CONDITIONAL ATTACH CONSOLE

Conditionally Attach to the
Default Virtual 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 virtual console of the calling process only if the virtual console is currently unattached.

If the virtual console is currently attached to another process, the function returns a value of 0FFH indicating that the virtual console could not be attached. The function returns a value of 0 to indicate that either the virtual console is already attached to the process or that it was unattached and a successful attach operation was made.

See Appendix M for a list of error codes returned in CX.

## FUNCTION 163: RETURN CCP/M VERSION NUMBER

Return the version of current
Concurrent CP/M-86 system

```
Entry Parameters:
    Register  CL: 0A3H

Return Values:
              AX: Version Number (01410H)
              BX: Same as AX
              CX: Error Code
```

The Return CCP/M Version Number function provides information that allows version independent programming. The function returns a two-byte value, with AH set to 014H for Concurrent CP/M-86, and AL set to the Concurrent CP/M-86 version level. A value of 01410H indicates Concurrent CP/M-86 1.0.

See Appendix M for a list of error codes returned in CX.

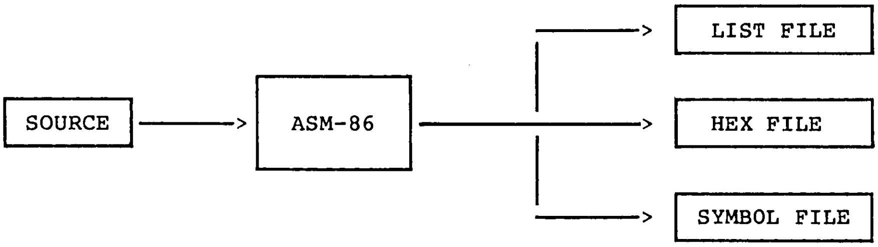
```
FUNCTION 164:  GET LIST NUMBER

        Return the Calling Process'
            Default List Device

Entry Parameters:
    Register  CL: 0A4H

Return Values:
              AL: List Device Number
              BL: Same as AL
```

The Get List Number function returns the default list device number of the calling process.

# Section 6
# Introduction to ASM-86

## 6.1 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 can 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 the Intel 8080 or the Zilog Z80® based system, and a 8086 assembler designed to run under Concurrent CP/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 6-1, below.



| | | |
|---|---|---|
| <filename>.A86 | - | contains source |
| <filename>.LST | - | contains listing |
| <filename>.H86 | - | contains assembled program in hexadecimal format |
| <filename>.SYM | - | contains all user-defined symbols |

**Figure 6-1. ASM-86 Source and Object Files**

Figure 6-1 also lists ASM-86 filetypes. ASM-86 accepts a source file with any three-letter extension, but if the filetype is omitted from the starting command, ASM-86 looks for the specified filename with the filetype .A86 in the directory. If the file has a filetype other than .A86 or has no filetype at all, ASM-86 returns an error message.

The other filetypes listed in Figure 6-1 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.

Start ASM-86 by entering a command of the following form:

```
ASM86  <source filename> [ $ <optional parameters> ]
```

Section 6.2 explains the optional parameters. Specify the source file in the following form:

```
[<optional drive>:]<filename>[.<optional filetype>]
```

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 filetype> | is a valid filetype of 1 to 3 characters, usually .A86. |

Some examples of valid ASM-86 commands are:

```
0A>ASM86 B:BIOS88

0A>ASM86 BIOS88.A86  $FI AA HB PB SB

0A>ASM86 D:TEST
```

Once started, 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 filetype 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 can be redirected by optional parameters or by a drive specification in the the source filename. In the latter case, ASM-86 directs the output files to the drive specified in the source filename.

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
```

## 6.2 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 6-1, below.

**Table 6-1. 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 can 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 can be temporarily stopped at any time by typing a CTRL-S. Restart the output by typing a second CTRL-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 details these formats. If the F parameter is not entered in the command line, ASM-86 produces Digital Research hex format.

**Table 6-2. Run-time Parameter Examples**

| Command Line | Result |
|---|---|
| ASM86 IO | Assemble file IO.A86, produce IO.H86, 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.H86, no symbol file. |
| ASM86 IO $ PY SX | Assemble file IO.A86, produce IO.H86, route listing directly to printer, output symbols on console. |
| ASM86 IO $ FD | Produce Digital Research hex format. |
| ASM86 IO $ FI | Produce Intel hex format. |

## 6.3 Aborting ASM-86

You can abort ASM-86 execution at any time by pressing 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.

End of Section 6

# Section 7
# Elements of ASM-86 Assembly Language

## 7.1 ASM-86 Character Set

ASM-86 recognizes a subset of the ASCII character set. The valid characters are the alphanumerics, special characters, and nonprinting 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 can appear in a string.

## 7.2 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 can 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.

## 7.3 Delimiters

Delimiters mark the end of a token and add special meaning to the instruction; separators merely mark the end of a token. When a delimiter is present, separators need not be used. However, using separators after delimiters can make your program easier to read.

Table 7-1 describes ASM-86 separators and delimiters. Some delimiters are also operators and are explained in greater detail in Section 7.6.

**Table 7-1. 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 in 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, allowing multiple statements on a single source line |
| ' | apostrophe | delimits string constants |

## 7.4 Constants

A constant is a value known at assembly time that does not change while the assembled program is executed. A constant can be either an integer or a character string.

### 7.4.1 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 7-2, below.

Table 7-2. 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 can 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 0s and 1s. Octal digits range from 0 to 7; decimal digits range from 0 to 9. Hexadecimal constants contain decimal digits and 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 a leading 0, to prevent this problem. The following are valid numeric constants:

```
1234     1234D    1100B    1111000011110000B
1234H    0FFEH    33770    13772Q
33770    0FE3H    1234d    0ffffh
```

### 7.4.2 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 a 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 in character strings, so both upper- and lower-case letters can be used. Note that only alphanumerics, special characters, and spaces are allowed in strings.

A DB assembler directive is the only ASM-86 statement that can contain strings longer than two characters. The string cannot exceed 255 bytes. Include any apostrophe you want printed in the string by entering it twice. ASM-86 interprets the two keystrokes '' as a single apostrophe. Table 7-3 shows valid strings and how they appear after processing:

**Table 7-3. 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 |

## 7.5 Identifiers

Identifiers are character sequences that have symbolic meaning to the assembler. All identifiers in ASM-86 must obey the following rules:

- The first character must be alphabetic (A,...Z, a,...z).
- 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.
- Identifiers can be of any length up to the limit of the physical line.

There are two types of identifiers. The first are keywords that have predefined meanings to the assembler. The second are symbols that are defined by the user. The following are all valid identifiers:

```
NOLIST
WORD
AH
Third_street
How_are_you_today
variable@number@1234567890
```

### 7.5.1 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 8. Section 7.6 defines operators. Table 7-4 lists the ASM-86 keywords that identify 8086 registers.

Three keywords, BYTE, WORD, and DWORD, are predefined numbers. 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.

**Table 7-4. Register Keywords**

| Register Symbol | Size | Numeric Value | Meaning |
|---|---|---|---|
| AH | 1 byte | 100 B | Accumulator-High-Byte |
| BH | 1 byte | 111 B | Base-Register-High-Byte |
| CH | 1 byte | 101 B | Count-Register-High-Byte |
| DH | 1 byte | 110 B | Data-Register-High-Byte |
| AL | 1 byte | 000 B | Accumulator-Low-Byte |
| BL | 1 byte | 011 B | Base-Register-Low-Byte |
| CL | 1 byte | 001 B | Count-Register-Low-Byte |
| DL | 1 byte | 010 B | Data-Register-Low-Byte |
| AX | 2 bytes | 000 B | Accumulator (full word) |
| BX | 2 bytes | 011 B | Base-Register " |
| CX | 2 bytes | 001 B | Count-Register " |
| DX | 2 bytes | 010 B | Data-Register " |
| BP | 2 bytes | 101 B | Base Pointer |
| SP | 2 bytes | 100 B | Stack Pointer |
| SI | 2 bytes | 110 B | Source Index |
| DI | 2 bytes | 111 B | Destination Index |
| CS | 2 bytes | 01 B | Code-Segment-Register |
| DS | 2 bytes | 11 B | Data-Segment-Register |
| SS | 2 bytes | 10 B | Stack-Segment-Register |
| ES | 2 bytes | 00 B | Extra-Segment-Register |

### 7.5.2 Symbols and Their Attributes

A symbol is a user-defined identifier that has attributes specifying the kind of information the symbol represents. Symbols fall into three categories:

- variables
- labels
- numbers

Variables 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 can 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 8.2.) A segment can 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 can 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 define variables as these three types. (See Section 8.) For example, a variable is defined when it appears as the name for a storage directive:

```
VARIABLE  DB 0
```

A variable can 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 and offset.

Label segment and offset attributes are essentially the same as variable segment and offset attributes. A label is defined when it precedes an instruction. A colon : separates the label from instruction; for example:

```
LABEL:   ADD  AX,BX
```

A label can also appear as the name for an EQU directive referencing another label. For example:

```
LABEL   EQU   ANOTHER_LABEL
```

Numbers can also be defined as symbols. A number symbol is treated as though 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 7.6 describes operators and their effects on numbers and number symbols.

## 7.6 Operators

ASM-86 operators fall into the following categories: arithmetic, logical, and relational operators, segment override, variable manipulators, and creators. Table 7-5 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 7-5. 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 0s become 1s, 1s become 0s. | a = 16-bit number |

**Table 7-5. (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 that results from shifting a to left by an amount b. | a, b = number |
| a SHR b | returns the value that 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 7-5. (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 the value of which is the segment value of the variable or label a. | a = label \| variable |
| OFFSET a | creates a number the value of which 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 is 1, 2, or 4, respectively. | a = label \| variable |
| LENGTH a | creates a number the value of which 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 |

### 7.6.1 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 can 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 the offset of which 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 the offset of which 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:

```
001E 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 can 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 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 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
```

### 7.6.2 Operator Precedence

Expressions combine variables, labels, or numbers with operators. ASM-86 allows several kinds of expressions. (See Section 7.7.) This section defines the order in which operations are executed should more than one operator appear in an expression.

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 leftmost is evaluated first. Table 7-6 presents ASM-86 operators in order of increasing precedence.

Parentheses can override 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 7-6. 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 | +, - |

**Table 7-6. (continued)**

| Order | Operator Type | Operators |
|---|---|---|
| 8 | Segment override | <segment override>: |
| 9 | Variable manipulators, creators | SEG, OFFSET, PTR, TYPE, LENGTH, LAST |
| 10 | Parentheses/brackets | ( ), [ ] |
| 11 | Period and Dollar | ., $ |

## 7.7 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. Numbers can 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 can consist of a base register, an index register, or both 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]
```

## 7.8 Statements

Just as tokens in this assembly language correspond to words in English, statements are analogous to sentences. A statement tells ASM-86 what action to perform. Statements can be instructions or 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 !. ASM-86 treats these 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 9. 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 can 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 can be present, although the other fields can appear. ASM-86 mnemonics are defined in Section 10.

operand(s)
: An instruction mnemonic can require other symbols to represent operands to the instruction. Instructions can have zero, one, or two operands.

comment
: Any semicolon ; appearing outside a character string begins a comment. A comment ends with a carriage return. Comments improve the readability of programs. This field is optional.

ASM-86 directives are described in Section 8. 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 only for 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 8.

operand(s)
: Analogous to the operands for instruction mnemonics. Some directives, such as DB, DW, and DD, allow any operand; others have special requirements.

comment
: Exactly as defined for instruction statements.

End of Section 7

# Section 8
# Assembler Directives

## 8.1 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 7.8.

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.

## 8.2 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. These are addressed by the CS, DS, SS, and ES registers, respectively. 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 can 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.

### 8.2.1 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 in 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.

### 8.2.2 The DSEG Directive

```
DSEG    <numeric expression>
DSEG
DSEG    $
```

This directive specifies that the following statements belong to the Data Segment. The Data Segment 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.

### 8.2.3 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.

### 8.2.4 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 legal.

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.

## 8.3 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 can 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.

## 8.4 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 nonzero 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. IF directives can be nested to five lvels.

## 8.5 The INCLUDE Directive

```
INCLUDE    <filename>
```

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 can not be nested; a source file called by an INCLUDE directive can not contain another INCLUDE statement. If <filename> does not contain a file type, the filetype is assumed to be .A86. If no drive name is specified with <filename>, ASM-86 assumes the drive containing the source file.

## 8.6 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).

## 8.7 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 can not terminate 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 assigns 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
```

## 8.8 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 7.4.2. 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, can be added to the definition, but cannot 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
```

## 8.9 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
```

## 8.10 The DD Directive

```
[symbol] DD <address expression>[,<address 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 is stored 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
```

## 8.11 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
```

## 8.12 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 gives the byte attribute.

## 8.13 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
```

## 8.14 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'
```

## 8.15 The PAGESIZE Directive

```
        PAGESIZE   <numeric expression>
```

The PAGESIZE directive defines the number of lines to be included on each printout page. The default page size is 66.

## 8.16 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 page width is 120 unless the listing is routed directly to the terminal; then the default page width is 78.

## 8.17 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.

## 8.18 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.

## 8.19 The NOLIST and LIST Directives

```
NOLIST
LIST
```

The NOLIST directive blocks the printout of the following lines. Restart the listing with a LIST directive.

## 8.20 The IFLIST and NOIFLIST Directives

```
IFLIST
NOIFLIST
```

The NOIFLIST directive suppresses the printout of the contents of IF-ENDIF blocks that are not assembled. The IFLIST directive resumes printout of IF-ENDIF blocks.

End of Section 8

# Section 9
# The ASM-86 Instruction Set

## 9.1 Introduction

The ASM-86 instruction set includes all 8086 machine instructions. The general syntax for instruction statements is given in Section 7.7. 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; 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. In each table, the instructions are listed alphabetically. Table 9-1 shows the symbols used in the instruction-definition tables to define operand types.

**Table 9-1. Operand Type Symbols**

| Symbol | Operand Type |
|---|---|
| numb | any ADDRESS expression |
| numb8 | any ADDRESS 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 9-1. (continued)**

| Symbol | Operand Type |
|---|---|
| mem | any ADDRESS expression, with or without base- and/or index-addressing modes, such as:<br><br>variable<br>variable+3<br>variable[bx]<br>variable[SI]<br>variable[BX+SI]<br>[BX]<br>[BP+DI] |
| simpmem | any ADDRESS expression WITHOUT base- and index-addressing modes, such as:<br><br>variable<br>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 that evaluates to a label |
| lab8 | any label that is within +/- 128 bytes distance from the instruction |

The 8086 CPU has nine single-bit Flag registers that reflect the state of the CPU. The user cannot access these registers directly, but the user 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 9-2 to represent the nine Flag registers.

**Table 9-2. Flag Register Symbols**

| Symbol | Meaning |
|---|---|
| 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 |

## 9.2 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 9-3 that if acc = AL, a byte is transferred, but if acc = AX, a word is transferred.

**Table 9-3. 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 |
| LAH | | 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 9-3. (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 is 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. |

## 9.3 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 9-4 summarizes the effects of arithmetic instructions on flag bits. Table 9-5 defines arithmetic instructions. Table 9-6 defines logical and shift instructions.

**Table 9-4. Effects of Arithmetic Instructions on Flags**

| Flag Bit | Result |
|---|---|
| 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 9-5. 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 |

**Table 9-5. (continued)**

| Syntax | | Result |
|---|---|---|
| DEC | mem\|reg | subtract 1 from memory or register |
| INC | mem\|reg | add 1 to memory or register |
| DIV | mem\|reg | divide (unsigned) accumulator (AX or AL) by memory or register. If byte results, AL = quotient, AH = remainder. If word results, AX = quotient, DX = remainder |
| IDIV | mem\|reg | divide (signed) accumulator (AX or AL) by memory or register - quotient and remainder stored as in DIV |
| IMUL | mem\|reg | multiply (signed) memory or register by accumulator (AX or AL). If byte, results in AH, AL. If word, results in DX, AX. |
| MUL | mem\|reg | multiply (unsigned) memory or register by accumulator (AX or AL). Results stored as in IMUL. |
| NEG | mem\|reg | two's complement memory or register |
| SBB | reg,mem\|reg | subtract (with borrow) memory or register from register |
| SBB | mem\|reg,reg | subtract (with borrow) register from memory or register |
| SBB | mem\|reg,numb | subtract (with borrow) immediate data from memory or register |
| SUB | reg,mem\|reg | subtract memory or register from register |
| SUB | mem\|reg,reg | subtract register from memory or register |
| SUB | mem\|reg,numb | subtract data constant from memory or register |

**Table 9-6. 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 9-6. (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 9-6. (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 |

## 9.4 String Instructions

String instructions take one or two operands. The operands specify only the operand type, determining whether the 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 9-7. String Instructions**

| Syntax | | Result |
|---|---|---|
| CMPS | mem\|reg,mem\|reg | subtract source from destination, affect flags, but do not return result. |
| CMPSB | | an alternate mnemonic for CMPS which assumes a byte operand. |
| CMPSW | | an alternate mnemonic for CMPS which assumes a word operand. |
| LODS | mem\|reg | transfer a byte or word from the source operand to the accumulator. |
| LODSB | | an alternate mnemonic for LODS which assumes a byte operand. |
| LODSW | | an alternate mnemonic for LODS which assumes a word operand. |
| MOVS | mem\|reg,mem\|reg | move 1 byte (or word) from source to destination. |

**Table 9-7. (continued)**

| Syntax | | Result |
|---|---|---|
| MOVSB | | an alternate mnemonic for MOVS which assumes a byte operand. |
| MOVSW | | an alternate mnemonic for MOVS which assumes a word operand. |
| SCAS | mem\|reg | subtract destination operand from accumulator (AX or AL), affect flags, but do not return result. |
| SCASB | | an alternate mnemonic for SCAS which assumes a byte operand. |
| SCASW | | an alternate mnemonic for SCAS which assumes a word operand. |
| STOS | mem\|reg | transfer a byte or word from accumulator to the destination operand. |
| STOSB | | an alternate mnemonic for STOS which assumes a byte operand. |
| STOSW | | an alternate mnemonic for STOS which assumes a word operand. |

Table 9-8 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.

**Table 9-8. 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 |

## 9.5 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 can be absolute or it can depend upon a certain condition. Table 9-9 defines control transfer instructions. In the definitions of conditional jumps, above and below refer to the relationship between unsigned values. Greater than and less than refer to the relationship between signed values.

**Table 9-9. 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 |

**Table 9-9. (continued)**

| Syntax | | Result |
|---|---|---|
| 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 ) |
| 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 |

**Table 9-9. (continued)**

| Syntax | | Result |
|---|---|---|
| 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 |
| 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 ) |

**Table 9-9. (continued)**

| Syntax | | Result |
|---|---|---|
| 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 |
| 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 |

## 9.6 Processor Control Instructions

Processor control instructions manipulate the flag registers. Moreover, some of these instructions synchronize the 8086 CPU with external hardware.

**Table 9-10. 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 coprocessor). numb8 must be in the range 0, 63 |
| 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 can precede any other instruction. Buslock prevents coprocessors from gaining the bus; this is useful for shared-resource semaphores. |
| NOP | | no operation is performed |
| 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 10
# Code-Macro Facilities

## 10.1 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. A Code-Macro sends a bit stream to the output file, adding a new instruction to the assembler.

Because ASM-86 treats a Code-Macro as an instruction, you can start Code-Macros by using them as instructions in your program. The example below shows how to start MAC, an instruction defined by a Code-Macro.

```
          .
          .
          .
     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 that are replaced by the names or values supplied as operands when the Code-Macro starts. 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 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. Those that appear to duplicate ASM-86 directives (DB, DW, and DD) have different meanings in Code-Macro context. These directives are detailed 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
```

## 10.2 Specifiers

Every formal parameter must have a specifier letter that indicates the type of operand needed to match the formal parameter. Table 10-1 defines the eight possible specifier letters.

**Table 10-1. 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. |

## 10.3 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 10-2 summarizes Code-Macro modifiers.

**Table 10-2. 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 | | |

## 10.4 Range Specifiers

The optional range is specified in parentheses by one expression or by 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 ranges from 0 to 63, inclusive:

```
CodeMacro ESC opcode:Db(0,63),adds:Eb
```

## 10.5 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. Those that appear to duplicate assembly language instructions have different meanings in a Code-Macro definition. Only the nine directives defined here are legal in Code-Macro definitions.

### 10.5.1 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 that represents the memory address. Because it represents a memory address, the formal parameter must have one of the specifiers E, M, or X.

### 10.5.2 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, and 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
```

### 10.5.3 MODRM

This directive instructs the assembler to generate the ModRM byte that follows the opcode byte in many 8086 instructions. The ModRM byte contains either the indexing type or the register number to be used in the instruction. It also specifies the register 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 on 8086 instructions and their bit patterns, see Intel's 8086 Assembly Language Programming 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
```

### 10.5.4 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 supplied as an operand. RELB generates one byte and RELW two bytes of displacement. The directives take 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
```

### 10.5.5 DB, DW, and DD

These directives differ from those that occur outside of Code-Macros. The forms 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
```

### 10.5.6 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>))
```

<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 instructing the assembler to put a certain number in the specified position. This number usually 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 |

A <rshift>, 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, 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.

End of Section 10

# Section 11
# DDT-86

## 11.1 DDT-86 Operation

The DDT-86 program allows the user to test and debug programs interactively in a Concurrent CP/M-86 environment. You should be familiar with the 8086 processor, ASM-86, and the Concurrent CP/M-86 operating system.

### 11.1.1 Starting DDT-86

Start 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 the 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 filetype is omitted from the filename, .CMD is assumed. Note that DDT-86 cannot load a file of type .H86. The second form of the starting command is equivalent to the sequence

```
0A>DDT86
DDT86 x.x
-Efilename
```

At this point, the program that was loaded is ready for execution.

### 11.1.2 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 CTRL-C to end the debugging session. (See Section 11.1.4.) A command line can have up to 64 characters and must terminate with a carriage return. While entering the command, use standard CP/M line-editing functions (such as CTRL-X, CTRL-H, and CTRL-R) 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 11-1 summarizes DDT-86 commands. DDT-86 commands are defined individually in Section 11.2.

**Table 11-1. DDT-86 Command Summary**

| Command | Action |
|---|---|
| A | enter assembly language statements |
| B | compare blocks of memory |
| 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 |
| QI | read I/O port |
| QO | write I/O port |
| R | read disk file into memory |
| S | set memory to new values |
| SR | search for string |
| 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 can be followed by one or more arguments. These can be hexadecimal values, filenames, 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.

### 11.1.3 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 can 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 11.3.

### 11.1.4 Terminating DDT-86

Terminate DDT-86 by typing a CTRL-C in response to the hyphen prompt. This returns control to the CCP. Note that Concurrent CP/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.

### 11.1.5 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 initially starts, 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 started, 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.

## 11.2 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.

### 11.2.1 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 7.8. 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.

### 11.2.2 The B (Block Compare) Command

The B command compares two blocks of memory and displays any differences on the screen. The form is:

```
Bs1,f1,s2
```

where s1 is the 20-bit address of the start of the first block; f1 is the offset of the final byte of the first block, and s2 is the 20-bit address of the start of the second block. If the segment is not specified in s2, the same value is used that was used for s1.

Any differences in the two blocks are displayed at the screen in the following form:

```
s1:o1 b1     s2:o2 b2
```

where s1:o1 and s2:o2 are the addresses in the blocks; b1 and b2 are the values at the indicated addresses. If no differences are displayed, the blocks are identical.

### 11.2.3 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 nongraphic 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, you can abort the D command by typing any character at the console.

### 11.2.4 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 forms:

```
E<filename>
E
```

where <filename> is the name of the file to be loaded. If no filetype 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 can 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. Files are closed after an E command.

E with no <filename> frees all memory allocations made by DDT-86, without loading a file.

### 11.2.5 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 in 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 nonexistent RAM at the location indicated.

### 11.2.6 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.

### 11.2.7 The H (Hexadecimal Math) Command

The H command computes the sum and difference of two 16-bit values. The form is shown below:

```
Ha,b
```

where a and b are the values the sum and difference of which are being computed. DDT-86 displays the sum (ssss) and the difference (dddd) truncated to 16 bits on the next line, as shown below:

```
ssss dddd
```

### 11.2.8 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 16-bit value at absolute memory location 0:6. 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.

### 11.2.9 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 can be aborted by typing any key during the list process. Or enter CTRL-S to halt the display temporarily.

### 11.2.10 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.

### 11.2.11 The QI, QO (Query I/O) Commands

The QI and QO commands allow access to any of the 65,536 input/output ports. The QI command reads data from a port; the QO command writes data to a port. The forms of the QI command are:

```
QIn
QIWn
```

where n is the 16-bit port number. In the first case, DDT-86 displays the 8-bit value read from port n. In the second case, DDT-86 displays a 16-bit value from port n.

The forms of the QO command are:

```
QOn,v
QOWn,v
```

where n is the 16-bit port number, and v is the value to output. In the first case, the 8-bit value v is written to port n. If v is greater than 255, DDT-86 responds with a question mark. In the second case, the 16-bit value v is written to port n.

### 11.2.12 The R (Read) Command

The R command reads a file into a contiguous block of memory. The forms are:

```
R<filename>
R<filename>,s
```

where <filename> is the name and type of the file to be read, and s is the location to which the file is read. The first form lets DDT-86 determine the memory location into which the file is read. The second form tells DDT-86 to read the file into the memory segment beginning at s. This address can have the standard form (ssss:oooo). The low-order four bits of s are assumed to be zero, so DDT-86 reads files on a paragraph boundary. If the memory at s is not available, DDT-86 issues the message:

```
MEMORY REQUEST DENIED
```

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 (f or 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 can be read into memory without overlapping.

If the file does not exist or there is not enough memory to load the file, DDT-86 issues an error message. Files are closed after an R command, even if an error occurs.

Examples:

| | |
|---|---|
| rddt86.cmd | Read file DDT86.CMD into memory. |
| rtest | Read file TEST into memory. |
| rtest,1000:0 | Read file TEST into memory, starting at location 1000:0. |

### 11.2.13 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
```

In response to the second form, the display is

```
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 can 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 nonexistent RAM at the location indicated.

### 11.2.14 The SR (Search) Command

The SR (Search) command searches a block of memory for a given pattern of numeric or ASCII values and lists the addresses where the pattern occurs. The form is:

```
SRs,f,<pattern>
```

where s is the 20-bit starting address of the block to be searched, f is the offset of the final address of the block, and <pattern> is a list of one or more hexadecimal values and/or ASCII strings. ASCII strings are enclosed in double quotes and can be of any length.

Example:

```
SR200,1000,"The form",0d,0a
```

For each occurrence of <pattern>, DDT-86 displays the 20-bit address of the first byte of the pattern, in the form

```
ssss:oooo
```

If no addresses are listed, <pattern> was not found.

### 11.2.15 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, allowing 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, forcing 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 can 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 because 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.

### 11.2.16 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 can be aborted before n steps have been executed by striking any key at the console.

### 11.2.17 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.

### 11.2.18 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 filetype 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.

### 11.2.19 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 can be 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 11-2. <instruction> is the disassembled instruction at the next location to be executed, indicated by the CS and IP registers.

**Table 11-2. 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 until 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 can be examined or altered with each Xf command. Set or reset flags by entering a value of 1 or 0.

After an X command, the typel and type2 segment values are set to the contents of the CS and DS registers, respectively.

## 11.3 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 started, 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.
- After an X command, the typel and type2 segment values are set to the contents of the CS and DS registers, respectively.
- 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 started, 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 ofthe 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 because it defaults to the CS register.

Table 11-3 summarizes DDT-86's default segment values.

**Table 11-3. DDT-86 Default Segment Values**

| Command | type-1 | type-2 |
|---|---|---|
| A | x | |
| B | | 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

## 11.4 Assembly Language Syntax for A and L Commands

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) can appear in one statement, but they all must precede the opcode of the statement. Alternately, a prefix can 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 either to a byte or word are ambiguous and must be preceded by either the prefix BYTE or WORD. These prefixes can be abbreviated BY and WO. For example:

  ```
  INC     BYTE [BP]
  NOT     WORD [1234]
  ```

  Failure to supply a prefix when needed results in an error message.

- Operands that 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 can be preceded by a numeric offset. For example:

  ```
  ADD     BX,[BP+SI]
  ADD     BX,3[BP+SI]
  ADD     BX,1D47[BP+SI]
  ```

## 11.5 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
:1B000081BE0000BB0001C6060801008A003A4001760A8640018800C60608016C
:11001B81014683FE0875E9F60608010175D7E9FDFFEE
:09010082030804061F0604010035
: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 ................

          Interesting, the fourth element seems to have been sorted in.
-esort
     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.*
*Reload program.*

```
-esort
     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.
-180
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 filetype are optional) |
| <parameters> | are a one-letter type followed by a one-letter device from the table below. |

Default filetype: .A86

Parameters:

form: $ Td where T = type and d = device

**Table A-1. Parameter Types and Devices**

| TYPES:<br>DEVICES: | A | H | P | S | F |
|---|---|---|---|---|---|
| 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**

| Type | Function |
|---|---|
| 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**

| Name | Meaning |
|---|---|
| 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**

| Example | Result |
|---|---|
| ASM86 IO | Assemble file IO.A86, produce IO.H86 IO.LST and IO.SYM. |
| ASM86 IO.ASM $ AD SZ | Assemble file IO.ASM on device D, produce IO.LST and IO.H86, no symbol file. |
| ASM86 IO $ PY SX | Assemble file IO.A86, produce IO.H86, route listing directly to printer, output symbols on console. |
| ASM86 IO $ FD | Produce Digital Research hex format. |
| ASM86 IO $ FI | Produce Intel hex format. |

End of Appendix A

# 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 |
|---|---|---|
| Intrasegment short jump: | JMPS | JMP |
| Intersegment jump: | JMPF | JMP |
| Intersegment return: | RETF | RET |
| Intersegment call: | CALLF | CALL |

End of Appendix B

# 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 Digital 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**

| 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 hexadecimal format differs from Intel. 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 Value | Digital 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 |

End of Appendix C

# 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 |
| IFLIST | NOIFLIST | | | |
| **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. | | | | |

End of Appendix D

# Appendix E
# ASM-86 Instruction Summary

**Table E-1. ASM-86 Instruction Summary**

| Mnemonic | Description | Section |
|---|---|---|
| AAA | ASCII adjust for Addition | 9.3 |
| AAD | ASCII adjust for Division | 9.3 |
| AAM | ASCII adjust for Multiplication | 9.3 |
| AAS | ASCII adjust for Subtraction | 9.3 |
| ADC | Add with Carry | 9.3 |
| ADD | Add | 9.3 |
| AND | And | 9.3 |
| CALL | Call (intrasegment) | 9.5 |
| CALLF | Call (intersegment) | 9.5 |
| CBW | Convert Byte to Word | 9.3 |
| CLC | Clear Carry | 9.6 |
| CLD | Clear Direction | 9.6 |
| CLI | Clear Interrupt | 9.6 |
| CMC | Complement Carry | 9.6 |
| CMP | Compare | 9.3 |
| CMPS | Compare Byte or Word (of string) | 9.4 |
| CMPSB | Compare Byte of string | 9.4 |
| CMPSW | Compare Word of string | 9.4 |
| CWD | Convert Word to Double Word | 9.3 |
| DAA | Decimal Adjust for Addition | 9.3 |
| DAS | Decimal Adjust for Subtraction | 9.3 |
| DEC | Decrement | 9.3 |
| DIV | Divide | 9.3 |
| ESC | Escape | 9.6 |
| HLT | Halt | 9.6 |
| IDIV | Integer Divide | 9.3 |
| IMUL | Integer Multiply | 9.3 |
| IN | Input Byte or Word | 9.2 |
| INC | Increment | 9.3 |
| INT | Interrupt | 9.5 |
| INTO | Interrupt on Overflow | 9.5 |
| IRET | Interrupt Return | 9.5 |
| JA | Jump on Above | 9.5 |
| JAE | Jump on Above or Equal | 9.5 |
| JB | Jump on Below | 9.5 |
| JBE | Jump on Below or Equal | 9.5 |
| JC | Jump on Carry | 9.5 |
| JCXZ | Jump on CX Zero | 9.5 |
| JE | Jump on Equal | 9.5 |
| JG | Jump on Greater | 9.5 |
| JGE | Jump on Greater or Equal | 9.5 |
| JL | Jump on Less | 9.5 |
| JLE | Jump on Less or Equal | 9.5 |

**Table E-1. (continued)**

| Mnemonic | Description | Section |
|---|---|---|
| JMP | Jump (intrasegment) | 9.5 |
| JMPF | Jump (intersegment) | 9.5 |
| JMPS | Jump (8-bit displacement) | 9.5 |
| JNA | Jump on Not Above | 9.5 |
| JNAE | Jump on Not Above or Equal | 9.5 |
| JNB | Jump on Not Below | 9.5 |
| JNBE | Jump on Not Below or Equal | 9.5 |
| JNC | Jump on Not Carry | 9.5 |
| JNE | Jump on Not Equal | 9.5 |
| JNG | Jump on Not Greater | 9.5 |
| JNGE | Jump on Not Greater or Equal | 9.5 |
| JNL | Jump on Not Less | 9.5 |
| JNLE | Jump on Not Less or Equal | 9.5 |
| JNO | Jump on Not Overflow | 9.5 |
| JNP | Jump on Not Parity | 9.5 |
| JNS | Jump on Not Sign | 9.5 |
| JNZ | Jump on Not Zero | 9.5 |
| JO | Jump on Overflow | 9.5 |
| JP | Jump on Parity | 9.5 |
| JPE | Jump on Parity Even | 9.5 |
| JPO | Jump on Parity Odd | 9.5 |
| JS | Jump on Sign | 9.5 |
| JZ | Jump on Zero | 9.5 |
| LAHF | Load AH with Flags | 9.2 |
| LDS | Load Pointer into DS | 9.2 |
| LEA | Load Effective Address | 9.2 |
| LES | Load Pointer into ES | 9.2 |
| LOCK | Lock Bus | 9.6 |
| LODS | Load Byte or Word (of string) | 9.4 |
| LODSB | Load Byte of string | 9.4 |
| LODSW | Load Word of string | 9.4 |
| LOOP | Loop | 9.5 |
| LOOPE | Loop While Equal | 9.5 |
| LOOPNE | Loop While Not Equal | 9.5 |
| LOOPNZ | Loop While Not Zero | 9.5 |
| LOOPZ | Loop While Zero | 9.5 |
| MOV | Move | 9.2 |
| MOVS | Move Byte or Word (of string) | 9.4 |
| MOVSB | Move Byte of string | 9.4 |
| MOVSW | Move Word of string | 9.4 |
| MUL | Multiply | 9.3 |
| NEG | Negate | 9.3 |
| NOT | Not | 9.3 |
| OR | Or | 9.3 |
| OUT | Output Byte or Word | 9.2 |

**Table E-1. (continued)**

| Mnemonic | Description | Section |
|---|---|---|
| POP | Pop | 9.2 |
| POPF | Pop Flags | 9.2 |
| PUSH | Push | 9.2 |
| PUSHF | Push Flags | 9.2 |
| RCL | Rotate through Carry Left | 9.3 |
| RCR | Rotate through Carry Right | 9.3 |
| REP | Repeat | 9.4 |
| RET | Return (intrasegment) | 9.5 |
| RETF | Return (intersegment) | 9.5 |
| ROL | Rotate Left | 9.3 |
| ROR | Rotate Right | 9.3 |
| SAHF | Store AH into Flags | 9.2 |
| SAL | Shift Arithmetic Left | 9.3 |
| SAR | Shift Arithmetic Right | 9.3 |
| SBB | Subtract with Borrow | 9.3 |
| SCAS | Scan Byte or Word (of string) | 9.4 |
| SCASB | Scan Byte of string | 9.4 |
| SCASW | Scan Word of string | 9.4 |
| SHL | Shift Left | 9.3 |
| SHR | Shift Right | 9.3 |
| STC | Set Carry | 9.6 |
| STD | Set Direction | 9.6 |
| STI | Set Interrupt | 9.6 |
| STOS | Store Byte or Word (of string) | 9.4 |
| STOSB | Store Byte of string | 9.4 |
| STOSW | Store Word of string | 9.4 |
| SUB | Subtract | 9.3 |
| TEST | Test | 9.3 |
| WAIT | Wait | 9.6 |
| XCHG | Exchange | 9.2 |
| XLAT | Translate | 9.2 |
| XOR | Exclusive Or | 9.3 |

End of Appendix E

# Appendix F
# Sample Program

```
                       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    *
                       ;       **********************
```

```
CP/M ASM86  1.09   SOURCE:  APPF.A86        Terminal  Input/Output
PAGE   2


                           ;
                           ;         Terminal 1:
                           ;
   0010                    instat1         equ     10h      ; input status port
   0011                    indata1         equ     11h      ; input port
   0011                    outdata1        equ     11h      ; output port
   0001                    readyinmask1    equ     01h      ; input ready mask
   0002                    readyoutmask1   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
                           constat1:
 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
PAGE    3


                       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
```

```
CP/M ASM86 1.09   SOURCE: APPF.A86        Terminal Input/Output
PAGE   4


 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 *
                    ;       *************************
```

```
CP/M ASM86 1.09   SOURCE: APPF.A86         Terminal Input/Output
PAGE   5


                        ;

 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
```

End of Appendix F

# 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
```

End of Appendix G

# Appendix H
# ASM-86 Error Messages

ASM-86 produces two types of error messages: fatal errors and diagnostics. Fatal errors occur when ASM-86 is unable to continue assembling. Diagnostics messages report problems with the syntax and semantics of the program being assembled. The following messages indicate fatal errors ASM-86 encounters during assembly:

```
NO FILE
DISKETTE FULL
DIRECTORY FULL
DISKETTE 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 CODEMACRO BUILDING |

End of Appendix H

# 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 can 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 nonexistent memory at the indicated location. |

End of Appendix I

# Appendix J
# System Function Summary

**Table J-1. System Function Summary**

| Dec | Hex | Function Name | Input Parameters | Returned values |
|---|---|---|---|---|
| 0 | 0 | System Reset | none | none |
| 1 | 1 | Con Input | none | AL = char |
| 2 | 2 | Con Output | DL = char | none |
| 3 | 3 | Raw Con Input | none | AL = char |
| 4 | 4 | Raw Con Output | DL = char | none |
| 5 | 5 | List Output | DL = char | none |
| 6 | 6 | Direct Con I/O | see def | see def |
| 7 | 7 | Get I/O Byte | ** Not supported in Concurrent CP/M-86 ** | |
| 8 | 8 | Set I/O Byte | ** Not supported in Concurrent CP/M-86 ** | |
| 9 | 9 | Print String | DX = .Buffer | none |
| 10 | A | Read Con Buffer | DX = .Buffer | see def |
| 11 | B | Get Con Status | none | AL = 00/01 |
| 12 | C | Rtn Version # | none | AL= Version# |
| 13 | D | Reset Disk Sys | none | see def |
| 14 | E | Select Disk | DL = Disk Number | see def |
| 15 | F | Open File | DX = .FCB | AL = Dir Code |
| 16 | 10 | Close File | DX = .FCB | AL = Dir Code |
| 17 | 11 | Search for First | DX = .FCB | AL = Dir Code |
| 18 | 12 | Search for Next | none | AL = Dir Code |
| 19 | 13 | Delete File | DX = .FCB | AL = Dir Code |
| 20 | 14 | Read Sequential | DX = .FCB | AL = Err Code |
| 21 | 15 | Write Sequential | DX = .FCB | AL = Err Code |
| 22 | 16 | Make File | DX = .FCB | AL = Dir Code |
| 23 | 17 | Rename File | DX = .FCB | AL = Dir Code |
| 24 | 18 | Rtn Login Vect | none | AX = Login Vect* |
| 25 | 19 | Rtn Current Disk | none | AX = Cur Disk# |
| 26 | 1A | Set DMA Address | DX = .DMA | none |
| 27 | 1B | Get Addr(Alloc) | none | AX = .Alloc |
| 28 | 1C | Write Prot Disk | none | see def |
| 29 | 1D | Get R/O Vect | none | AX = R/O Vect* |
| 30 | 1E | Set File Attrib | DX = .FCB | see def |
| 31 | 1F | Get Addr(Disk parms) | none | AX = .DPB |
| 32 | 20 | Set/Get User Code | see def | see def |
| 33 | 21 | Read Random | DX = .FCB | AL = Err Code |
| 34 | 22 | Write Random | DX = .FCB | AL = Err Code |
| 35 | 23 | Compute File Size | DX = .FCB | r0, r1, r2 |
| 36 | 24 | Set Random Rec | DX = .FCB | r0, r1, r2 |
| 37 | 25 | Reset Drive | DX = drive Vect | AL = Err Code |
| 38 | 26 | Access Drive | DS = drive Vect | none |
| 39 | 27 | Free Drive | DS = drive Vect | none |
| 40 | 28 | Write Random 0-fill | DS = .FCB | AL = Err Code |
| 41 | 29 | Test/Write Rec | DS = .FCB | AL = Err Code |

**Table J-1. (continued)**

| Dec | Hex | Function Name | Input Parameters | Returned values |
|---|---|---|---|---|
| 42 | 2A | Lock Rec | DS = .FCB (Current DMA Addr | AL = Err Code -> File ID) |
| 43 | 2B | Unlock Rec | DX = .FCB (Current DMA Addr | AL = Err Code -> File ID) |
| 44 | 2C | Set Multi-Sector Ct | DL= # of Sectors | AL = Rtn Code |
| 45 | 2D | Set BDOS Err Mode | see def | none |
| 46 | 2E | Get Disk Free Sp | DL = Disk # | see def |
| 47 | 2F | Chain To Program | see def | none |
| 48 | 30 | Flush Buffers | none | see def |
| 50 | 32 | Direct BIOS Call | DX = BD Addr. | AX = BIOS rtn |
| 51 | 33 | Set DMA Base | DX = DMA Seg.Addr | none |
| 52 | 34 | Get DMA Base | none | AX = DMA Offset |
| 53 | 35 | Get Max Mem | DX = MCB Addr | see def |
| 54 | 36 | Get Abs Max | DX = MCB Addr | see def |
| 55 | 37 | Alloc Mem | DX = MCB Addr | see def |
| 56 | 38 | Alloc Abs Max | DX = MCB Addr | see def |
| 57 | 39 | Free Mem | DX = MCB Addr | see def |
| 58 | 3A | Free All Mem | none | none |
| 59 | 3B | Program Load | DX = FCB Addr | AX = B.P.Seg |
| 100 | 64 | Set Dir Label | DX = .FCB | AL = Dir Code |
| 101 | 65 | Rtn Dir Label | DX = Disk # | AL = Label Data |
| 102 | 66 | Read File XFCB | DX = .XFCB | AL = Dir Code |
| 103 | 67 | Write File XFCB | DX = .XFCB | AL = Dir Code |
| 104 | 68 | Set Date/Time | DX = .TOD | none |
| 105 | 69 | Get Date/Time | DX = .TOD | none |
| 106 | 6A | Set Default Pswd | DX = .Password | none |
| 107 | 6B | Rtn Serial # | DX = .serialnmb | serialnmb set |
| 128 | 80 | Abs Memory Rqst | DX = .MD | AX = Rtn Code |
| 129 | 81 | Reloc Mem Rqst | DX = .MD | AX = Rtn Code |
| 130 | 82 | Memory Free | DX = .MD | none |
| 131 | 83 | Poll | DL = Device | none |
| 132 | 84 | Flag Wait | DL = Flag | AX = Rtn Code |
| 133 | 85 | Flag Set | DL = Flag | AX = Rtn Code |
| 134 | 86 | Make Queue | DX = QD addr | none |
| 135 | 87 | Open Queue | DX = QPB Addr | AX = Rtn Code |
| 136 | 88 | Delete Queue | DX = QPB Addr | AX = Rtn Code |
| 137 | 89 | Read Queue | DX = QPB Addr | none |
| 138 | 8A | Cond. Read Queue | DX = QPB Addr | AX = Rtn Code |
| 139 | 8B | Write Queue | DX = QPB Addr | none |
| 140 | 8C | Cond. Write Queue | DX = QPB Addr | AX = Rtn Code |
| 141 | 8D | Delay | DX = #ticks | none |
| 142 | 8E | Dispatch | none | none |
| 143 | 8F | Term. Proc | DL = Term. Code | none |
| 144 | 90 | Create Proc | DX = PD Addr | none |
| 145 | 91 | Set Priority | DL = Priority | none |
| 146 | 92 | Attach Con | none | none |
| 147 | 93 | Detach Con | none | none |
| 148 | 94 | Set Con | DL = Console | none |
| 149 | 95 | Assign Con | DX = ACB Addr | AX = Rtn Code |
| 150 | 96 | Send CLI Comm | DX = CLBUF Addr | none |

**Table J-1. (continued)**

| Dec | Hex | Function Name | Input Parameters | Returned values |
|---|---|---|---|---|
| 151 | 97 | Call RPL | DX = CPB Addr | AX = result |
| 152 | 98 | Parse Filename | DX = PFCB Addr | see def |
| 153 | 99 | Get Con # | none | AL = con # |
| 154 | 9A | Sys Data Addr | none | AX = Sys Data Addr |
| 155 | 9B | Get Date/Time | DX = TOD Addr | none |
| 156 | 9C | Rtn PD Addr | none | AX = PD Addr |
| 157 | 9D | Abort Spec. Proc | DX = ABP Addr | AL = Rtn Code |
| 158 | 9E | Attach List | none | none |
| 159 | 9F | Detach List | none | none |
| 160 | A0 | Set List | DL = List # | none |
| 161 | A1 | Cond. Attach List | none | AX = Rtn Code |
| 162 | A2 | Cond. Attach Con | none | AX = Rtn Code |
| 163 | A3 | MPM Version # | none | AX = Version # |
| 164 | A4 | Get List # | none | AL = list # |

The following abbreviations are used in the table.

| | | |
|---|---|---|
| Abs | = | Absolute |
| Addr | = | Address |
| Char | = | ASCII Character |
| Comm | = | Command |
| Con | = | Console |
| Cond. | = | Conditional |
| Ct | = | Count |
| Dir | = | Directory |
| Err | = | Error |
| Proc | = | Process |
| Pswd | = | Password |
| Reloc | = | Relocatable |
| Rec | = | Record |
| Rqst | = | Request |
| Rtn | = | Return |
| Sp | = | Space |
| Spec. | = | Specified |
| Sys | = | System |
| Term. | = | Terminate |
| Vect | = | Vector |
| # | = | Number |

**Note:** DL is the low-order half of register DX, and AL is the low-order half of register AX.

End of Appendix J

# Appendix K
# Glossary

**Base Page:** Memory region between 0000H and 0100H relative to the beginning of the Data Segment used to hold critical system parameters. Base Page functions primarily as an interface region between user programs and BDOS module. Note that in the 8080 Model, the code and data are intermixed in the code segment.

**BCD:** Acronym for Binary Coded Decimal. Representation of decimal numbers using binary digits. See Appendix L for binary representations of ASCII codes.

**block:** Basic unit of disk space allocation under Concurrent CP/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. 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 inadvertently switching disks without logging in the new disk. If not logged in, the new disk is treated the same as the old one, and data on it can be destroyed if writing is done.

**CMD:** Filetype for Concurrent CP/M-86 command files. These are machine language object modules ready to be loaded and executed. Any file with this type can be executed by simply typing the filename after the drive prompt. For example, the program PIP.CMD can be executed by simply typing PIP.

**command:** Set of instructions that are executed when the command name is typed after the system prompt. These instructions can be built in the Concurrent CP/M-86 system or can reside on disk as a file of type CMD. Concurrent CP/M-86 commands consist of three parts: the command name, the command tail, and a carriage return.

**console:** Primary I/O device used by Concurrent CP/M-86. The console usually consists of a CRT screen for displaying output and a keyboard for input.

**control character:** Nonprinting ASCII character produced on the console by holding down the CTRL (CONTROL) key while striking the character key. CTRL-H means hold down CTRL and hit H. Control characters are sometimes indicated using the up-arrow symbol (^), so CTRL-H can be represented as ^H. Certain control characters are treated as special commands by Concurrent CP/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 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 Concurrent CP/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 can have several elements, depending on its size. The maximum number of directory elements supported is specified in the drive's Disk Parameter Block.

**directory element:** 32-byte element associated with each disk file. A file can have more than one directory element associated with it. There are four directory elements per directory sector. Directory elements can also be referred to as directory FCBs.

**directory entry:** File entry displayed when using the DIR command. This term can also be used to refer to a physical directory element (FCB).

**disk, diskette:** Magnetic media used for mass storage of data in the computer system. The term disk can refer to a diskette, a removable cartridge disk, or a 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 Concurrent CP/M-86. The address of the DPB is in the Disk Parameter Header at DPbase + 0AH. Drives with the same characteristics can use the same diskette Parameter Header, and thus the same DPB. However, drives with different characteristics must each have their own Disk Parameter Header and DPB. The address of the drive's Disk Parameter Header must be returned in registers HL when the BDOS calls the SELDSK entry point in the BIOS. BDOS Function 31 returns the DPB address.

**Disk Parameter Header (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. The table's first row of 16 bytes corresponds to drive 0; the last row corresponds 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 can contain 1, 2, 4, 8, or 16 extents. The EX field is usually set to 0 by the user but contains the current extent number during file I/O. The term FCB Folding is used to describe FCBs 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):** Thirty-six 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 can be divided into one or more logical drives, and designated with specific drive references (such as d:a or d:f). Thus, at the user interface, it appears that there are several disks in the system.

**parse:** Separate a command line into its constituent parts.

**physical drive:** Peripheral hardware device used for mass storage of data within the computer system.

**Read-Only:** Condition in which a drive can be read but not written to. A drive can be set to Read-Only status by using the SET or STAT utilities. The only other way a drive can 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 can 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 diskette. If the blocking and deblocking algorithm is used, several records can 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; it must be pure code that does 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 referred to as Host Sectors. When the Host Sector size is larger than 128 bytes, Host Sectors must be buffered in memory, and the 128-byte sectors must be blocked and deblocked from them.

**source file:** ASCII text file usually created with a text editor that is an input file to a system program, such as a language translator or a text formatter.

**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.

**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 disks 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 ? or * characters. The BDOS directory search functions will match ? with any single character and * with multiple characters.

End of Appendix K

# Appendix L
# ASCII and Hexadecimal Conversions

This appendix contains tables of the ASCII symbols, including their binary, decimal, and hexadecimal conversions.

**Table L-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 L-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 L-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 L-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 |

End of Appendix L

# Appendix M
# Error Codes

**Table M-1. Concurrent CP/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 |

End of Appendix M

# Index