DIGITAL RESEARCH™

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

DIGITAL RESEARCH™

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

The *Concurrent CP/M-86 Operating System Programmer's Reference Guide* was printed in the United States of America.

First Edition: June 1983

Foreword

Concurrent CP/M-86™ is an operating system for 8086- or 8088-based microcomputer systems. It supports multiple CP/M® programming environments each implemented upon a virtual console. A different task runs concurrently in each environment.

This manual describes the invariant programming interface to Concurrent CP/M-86. It supports the applications programmer who must create applications programs that run in the CP/M-86® environment.

Section 1 offers an overview of the entire operating system.

Section 2 describes the structure of the Concurrent CP/M-86 file system.

Section 3 explains the format, structure, and uses of transient commands in the Concurrent CP/M-86 environment.

Section 4 explains the creation of transient command files in the Concurrent CP/M-86 environment.

Section 5 documents the structure and creation of resident system processes or resident command files permanently installed in the Concurrent CP/M-86 environment.

Section 6 describes all the Concurrent CP/M-86 system calls.

Concurrent CP/M-86 is supported and documented through four manuals:

- The *Concurrent CP/M-86 Operating System User's Guide* (hereafter cited as *Concurrent CP/M-86 User's Guide*) documents the user's interface to Concurrent CP/M-86, explaining the various features used to execute applications programs and Digital Research utility programs.

- The *Concurrent CP/M-86 Operating System Programmer's Reference Guide* (hereafter cited as *Concurrent CP/M-86 Programmer's Reference Guide*) documents the applications programmer's interface to Concurrent CP/M-86, explaining the internal file structure and system entry points, information that is essential for creating applications programs that run in the Concurrent CP/M-86 environment.
- The *Concurrent CP/M-86 Operating System Programmer's Utilities Guide* (hereafter cited as *Programmer's Utilities Guide*) is the Digital Research utility programs that programmers use to write, debug, and verify applications programs written for the Concurrent CP/M-86 environment.
- The *Concurrent CP/M-86 Operating System System Guide* (hereafter cited as *Concurrent CP/M-86 System Guide*) documents the internal, hardware-dependent structures of Concurrent CP/M-86.

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Appendixes

Table of Contents (continued)

Tables

Table of Contents (continued)

Tables

Figures

Table of Contents (continued)

Figures

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 initiating tasks on two or more virtual consoles. It is compatible with the CP/M-86 single-tasking operating system. Applications programs have access to system calls used by Concurrent CP/M-86 to control the multiprogramming environment. As a result, Concurrent CP/M-86 supports extended features, such as communication among and synchronization of independently running processes. Figure 1-1 depicts the relationships between applications programs, virtual environments, virtual consoles, and the physical console.

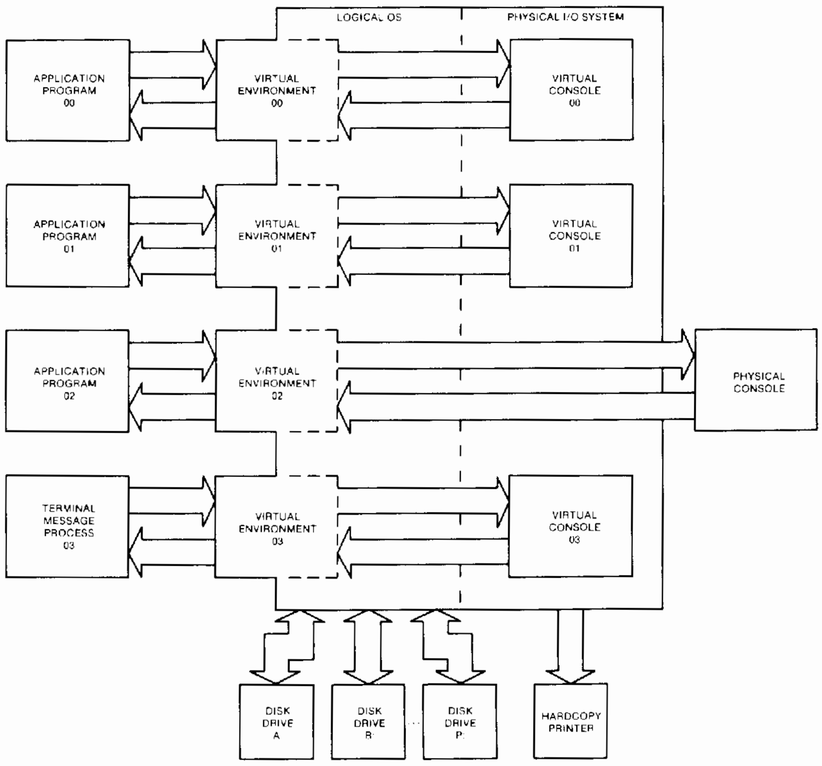

Figure 1-1. Concurrent CP/M-86 Virtual/Physical Environments

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 a dynamic entity. You can think of it as a logical machine that executes not only the program code, but also the operating system routines necessary to support the program's functions.

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 obtains access to the system's resources. Thus, Concurrent CP/M-86 monitors the process, not the program. This distinction is a subtle one, but vital to your understanding of system operation as a whole.

Processes running under Concurrent CP/M-86 fall into two categories: transient processes and Resident System Processes (RSPs). Transient processes run programs loaded into memory from disk in response to a user command or system calls made by another process. Resident system processes run code that is a part of the operating system itself. RSPs become an integral part of the operating system image during system generation. They are immediately available to perform operating system tasks. For example, the CLOCK process is an RSP that maintains the time of day within the operating system.

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

- Interprocess communication, synchronization, and mutual exclusion functions are provided by system queues.
- A logical interrupt mechanism using flags allows Concurrent CP/M-86 to interface with any physical interrupt structure.
- System timing functions enable processes running under Concurrent CP/M-86 to compute elapsed times, delay execution for specified intervals, and to access and set the current date and time.
- Shared file system allows multiple programs to access common data files while maintaining data integrity.

- Virtual console handling lets a single user run multiple programs, each in its own console environment.
- Real-time process control allows communications and data acquisition without loss of information.

Functionally, Concurrent CP/M-86 is composed of several distinct modules, as shown in Figure 1-2.

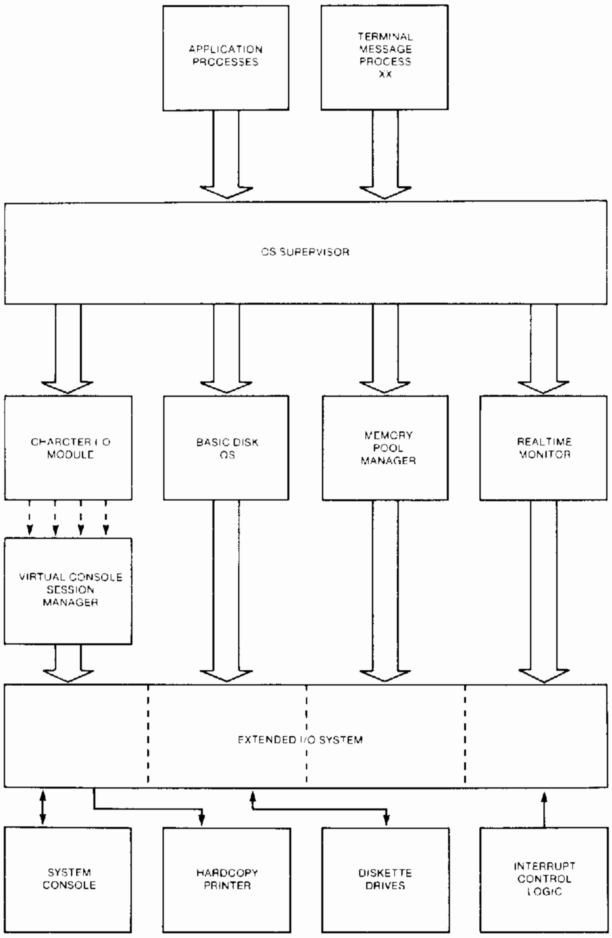

Figure 1-2. Concurrent CP/M-86 Functional Modules

- 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 calls such as returning the version number or the address of the System Data Area. SUP also calls other system calls when necessary.

The RTM module monitors the execution of running processes and arbitrates conflicts for the system's resources.

The MEM module allocates and frees memory upon demand from executing processes.

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

The 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 explained in detail in Section 2.

The XIOS is the hardware-dependent module that defines the interface of Concurrent CP/M-86 to a specific hardware environment. See the *Concurrent CP/M-86 System Guide* for an explanation of the XIOS.

When Concurrent CP/M-86 is executing a single program on a single virtual console, its speed approximates that of CP/M-86. But when multiple processes are running on several virtual consoles, the execution of each individual process slows according to the proportion of I/O to CPU 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 interface between processes and the operating system kernel. It also manages internal communication between operating system modules. All system calls, whether they originate from a transient process or internally from another system module, go through a common table-driven function interface in SUP. SUP also handles the P_LOAD (Load Process) and P_CLI (Call Command Line Interpreter) system calls.

1.3 Real-time Monitor (RTM)

The Real-time Monitor (RTM) is the real-time multitasking nucleus of Concurrent CP/M-86. The RTM performs process dispatching, queue management, flag management, device polling, and system timing tasks. User programs can also call many of the RTM system calls used to perform these tasks.

1.3.1 Process Dispatching

Although Concurrent CP/M-86 is a multiprocess operating system, only one process has access to the CPU resource at any given time. Unless you specifically write a program to communicate or synchronize execution with other processes, a process is unaware of other processes competing for system resources.

The primary task of the RTM is to transfer, or dispatch, the CPU resource from one process to another. The RTM module called the Dispatcher performs this task. The RTM maintains two data structures, the Process Descriptor (PD) and the User Data Area (UDA), for each process running under Concurrent CP/M-86. The Dispatcher uses these data structures to save and restore the current state of each running process.

Each process in the system resides in one of three states: ready, running, or suspended. A ready process is one that is waiting for the CPU resource only. A running process is one that the CPU is currently executing. A suspended process is one that is waiting for a system resource or a specified event, such as the occurrence of an interrupt, an indication that polled hardware is ready, or the expiration of a delay period.

Any existing process is represented on a system list. The Dispatcher removes a process from one list and places it on another. The Process Descriptor of the currently running process is the first entry on the Ready List. Other processes ready to run are represented on the Ready List in order of priority. Suspended processes are on other System Lists, depending on why the processes were suspended.

A dispatch operation can be summarized as follows:

1. The Dispatcher suspends the process from execution and stores its current state in the Process Descriptor and the UDA.
2. The Dispatcher places the process on an appropriate System List, depending on why the Dispatcher was called. For example, if a process is to delay for a certain number of system ticks, its Process Descriptor is placed on the Delay List. Where a process releases a resource, the process is usually placed back on the Ready List. If another process is waiting for the resource, that process is taken off its current System List and also placed on the Ready List.
3. The highest priority process on the Ready List is chosen for execution. If two or more processes have the same priority, the process that has waited the longest executes first.
4. The Dispatcher restores the state of the selected process from its Process Descriptor and UDA, and gives it the CPU resource.

5. The process executes until it needs a busy resource, a resource needed by another process becomes available, or an interrupt occurs. At this point, a dispatch occurs, allowing another process to run.

Only processes on the Ready List are eligible for selection during dispatch. By definition, a process is on the Ready List if it is waiting only for the CPU resource. Processes waiting for other system resources cannot execute until the resources they require are available. Concurrent CP/M-86 blocks a process from execution if it is waiting for:

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

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

A running process not needing a resource and not releasing one runs until an interrupt causes a dispatch. While not all interrupts cause dispatches, the system clock generates interrupts every clock tick and forces a dispatch each time. Clock ticks usually occur 60 times a second (approximately every 16.67 milliseconds), and allow time sharing within a real-time environment.

Concurrent CP/M-86 is a priority-driven system. This means that during a dispatch, the operating system gives the CPU resource to the process with the best priority. The Dispatcher allots equal shares of the system's resources to processes with the same priority. With priority dispatching, the system never passes control to a lower-priority process if there is a higher-priority process on the Ready List. Because high-priority, compute-bound processes tend to monopolize the CPU resource, it is best to reduce their priority to avoid degrading overall system performance.

1.3.2 Queue Management

Queues perform several critical functions for processes running under Concurrent CP/M-86. A process can use a queue for communicating with another process, synchronizing its execution with that of another process, and for exclusion of other processes from protected system resources. A process can make, open, delete, read from, or write to a queue with system calls similar to those used to manage disk files.

Each system queue consists of two parts: the queue descriptor, and the queue buffer. Concurrent CP/M-86 implements these special data structures as memory files that contain room for a specified number of fixed-length messages.

When the Q_MAKE system call creates a queue, this queue is assigned a unique 8-character name. As the name queue implies, messages are read from a queue on a first-in, first-out basis.

A process can read from or write to a queue conditionally or unconditionally. If the queue is empty when a conditional read is performed, or full when a conditional write is performed, the system returns an Error Code to the calling process. On the other hand, if a process attempts an unconditional queue operation in these circumstances, the system suspends it from execution until the operation becomes possible.

More than one process can wait to read or write a queue message from the same queue at the same time. When these operations become possible, the system restores the highest priority process first; processes with the same priority are restored 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 their names follow a convention, beginning with the upper-case letters MX. A mutual exclusion queue acts as a binary semaphore, ensuring that only one process uses a resource at any time.

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

1. A process issues an unconditional Q_READ call to the MX queue protecting the resource, thereby suspending itself if the message is not available.

2. When the message becomes available, the process accesses the protected resource. Note that from the time the process issues the unconditional read, any other process attempting to access the same resource is suspended.
3. The process writes the zero-length message back to the queue when it has finished using the protected resource, thus freeing the resource for other processes.

As an example, the system mutual exclusion queue, MXdisk, ensures that processes cannot access the file system simultaneously. Note that the BDOS, not the application software, executes the preceding series of queue calls. Therefore the mutual exclusion process is transparent to the programmer, who is only responsible for originating the disk system calls.

Mutual exclusion queues differ from normal queues in another way. When a process reads a message from a mutual exclusion queue, the RTM notes the Process Descriptor address within the Queue Descriptor. This establishes the owner of the queue message. If the operating system aborts the process 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 protected resources owned by the aborted process.

1.3.3 System Timing Functions

Concurrent CP/M-86's timing system calls 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 DEV_WAITFLAG system calls on the system's one second flag, Flag 2. When the XIOS Tick Interrupt Handler sets this flag, it initiates the CLOCK process, which then increments the internal time and date.

Subsequently, the CLOCK process makes another DEV_WAITFLAG call and suspends itself until the flag is set again. Concurrent CP/M-86 provides system calls that allow you to set and access the internal date and time. In addition, the file system uses the internal time and date to record when a file is updated, created, or last accessed.

The P_DELAY system call replaces the typical programmed delay loop for delaying process execution. P_DELAY requires that Flag 1, the system tick flag, be set approximately every 16.67 milliseconds, or 60 times a second; the XIOS Tick Interrupt Handler also sets this flag. When a process makes a P_DELAY system call, it specifies the number of ticks for which the operating system is to suspend it 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. When a DEV_SETFLAG call occurs, setting flag 1, the tick count is decremented. When the delay count goes to zero, the system removes the process from the Delay List and places it on the Ready List.

Note: the length of a tick might vary from installation to installation. For instance, in Europe, a tick is commonly 20 milliseconds, yielding 50 ticks per second. The description of the P_DELAY system call in Section 6 describes how to determine the correct number of ticks to delay 1 second.

1.4 Memory Module (MEM)

Concurrent CP/M-86 supports an extended, fixed partition model of memory management; the Memory Module handles all memory management system calls. In practice, the exact method that the operating system uses to allocate and free memory is transparent to the application program. Therefore you should take care to write code independent of the memory management model; use only the Concurrent CP/M-86-specific memory system calls described in Section 6.

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, facilitates 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. Concurrent CP/M-86's BDOS is extended to provide support for multiple virtual consoles and list devices. In addition, the file system is extended to provide services required in a multitasking environment. The major extensions to the file system are

- 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. The BDOS optionally supports two time and date stamps, one recording when a file is updated, and the other recording when the file was created or last accessed.
- Password Protection. The password protection feature is optional at either the file or drive level. The operator or applications program assigns disk drive passwords, while application programs can assign file protection passwords in several modes.
- Extended Error Module. Besides the default error mode, Concurrent CP/M-86 has two optional error-handling modes that return an error code to the calling process in the event of an irrecoverable disk error.

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). These data structures reside in the XIOS. The CCB contains the current owner, status information, line editing variables, and the root of a linked list of Process Descriptors (PDs) that are waiting for access. More than one process can wait for access to a single console. These processes are maintained on a linked list of Process Descriptors in priority order. The LCBs contain similar information about the list devices. See the *Concurrent CP/M-86 System Guide* for more information about LCBs and CCBs.

1.7 Virtual Console Screen Management

Virtual console screen management is coordinated by four separate modules: the CIO, the PIN (Physical INput) and VOUT (Virtual OUTput) processes, and the XIOS. The line editing associated with the C_READSTR call is performed in the CIO. The PIN process handles keyboard input for all the virtual consoles; it also traps and implements the CTRL-C, CTRL-S, CTRL-Q, CTRL-P, and CTRL-O functions. The VOUT process spools console output from processes running on background buffered mode consoles, and handshakes with the PIN process to display spooled console output when the background console is brought to the foreground. The XIOS decides which special keys represent the virtual consoles, and returns a special code from IO_CONIN when you request a screen switch. The XIOS also implements any screen saving and restoring when screens are switched. See the *Concurrent CP/M-86 System Guide* and the discussion of the IO_SWITCH function.

The PIN process reads the keyboard by directly calling the XIOS IO_CONIN function. This is the only place in the operating system IO_CONIN is called. The PIN scans the input stream from the keyboard for switch screen requests and the special function keystrokes CTRL-C, CTRL-S, CTRL-Q, CTRL-P, and CTRL-O. All other keyboard input is written to the VINQ (Virtual Console INput Queue) associated with the foreground virtual console. The data in the VINQ becomes a type-ahead buffer for each virtual console, and is returned to the process attached to that console as it performs console input.

When PIN sees a CTRL-C it calls P_ABORT to abort the process attached to the virtual console, flushes the type-ahead buffer in the VINQ, turns off CTRL-S, and performs a DRV_RESET call for each logged-in drive. The P_ABORT call succeeds when the Process Keep flag is not on, saving the Terminal Message Processes (refer to P_CREATE for information on the process descriptor). The DRV_RESET calls affect only the removable media drives, as specified in the CKS field of the Disk Parameter Blocks in the XIOS (refer to the *Concurrent CP/M-86 System Guide* for further details on Disk Parameter Blocks).

CTRL-S stops any output to the screen. CTRL-S stays set when a virtual console is switched to the background.

CTRL-O discards any console output to the virtual console. CTRL-O is turned off when any other key is subsequently pressed, except for the keys representing the virtual consoles.

CTRL-P echoes console output to the default list device specified in the LIST field of the process descriptor attached to the virtual console. If the list device is attached to a process, a PRINTER BUSY message appears.

All of the above control keys can be disabled by the C_MODE call. When one of the above control characters is disabled with C_MODE or when the process owning the virtual console is using the C_RAWIO call, the PIN does not act on the control character but instead writes it to the VINQ. It is thus possible to read any of the above control characters from an application program. These control keys are discussed in depth in the *Concurrent CP/M-86 User's Guide.*

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 operations, such as console I/O, are modified to support multiple virtual consoles. Several new primitive system calls, such as DEV_POLL, support Concurrent CP/M-86's additional features, including elimination of wait loops for real-time I/O operations.

1.9 Terminal Message Processes (TMP)

The Concurrent CP/M-86 Terminal Message Processes (TMPs) are resident system processes that accept command lines from the virtual consoles and call the Command Line Interpreter (CLI) 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 one that is not system-resident. The system must load such programs 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, loads it, and initiates it. Concurrent CP/M-86 supports three different execution models for transient programs: the 8080 Model, the Small Model, and the Compact Model. Sections 4.1.1 through 4.1.3 describe these models in detail.

1.11 System Call Calling Conventions

When a Concurrent CP/M-86 process makes a system call, it loads values into the registers shown in Table 1-1 and initiates Interrupt 224 (via the INT 224 instruction), reserved by the Intel® Corporation for this purpose.

Table 1-1. Registers Used by System Calls

	ENTRY PARAMETERS	
Register	CL:	System Call Number
	DL:	Byte Parameter
		or
	DX:	Word Parameter
		or
	DX:	Address - Offset
	DS:	Address - Segment
	RETURN VALUES	
Register	AL:	Byte Return
		or
	AX:	Word Return
		or
	AX:	Address - Offset
	ES:	Address - Segment
	BX:	Same as AX
	CX:	Error Code

Concurrent CP/M-86 preserves the contents of registers SI, DI, BP, SP, SS, DS, and CS through the operating system calls. The ES register is preserved when it is not used to hold a return segment value. Error codes returned in CX are shown in Table 6-5, CX Error Codes.

1.12 SYSTAT: System Status

The SYSTAT utility is a development tool that shows the internal state of Concurrent CP/M-86. SYSTAT describes memory allocation, current processes, system queue activity, and many informative parameters associated with these system data structures. Furthermore, SYSTAT presents two views: either a static snapshot of system activity, or a continuous, real-time window into Concurrent CP/M-86.

You can specify SYSTAT in one of two modes. If you know which display you want, you can specify it in the invocation, using an option shown in the menu below. If you do not specify an option, select a display from this menu by typing

```
A>SYSTAT <cr>
```

The screen clears and the main menu appears:

```
Which Option?

  H(elp)
  M(emory)
  O(verview)
  P(rocesses - All)
  Q(ueues)
  U(ser Processes)
  E(xit)

->_
```

Press the appropriate letter to obtain a display.

When you select H(elp), the HELP file demonstrates the proper syntax and available options:

```
To use SYSTAT with the menu: At the system prompt type SYSTAT <CR>

To use SYSTAT without the menu: At the system prompt type the command


              SYSTAT [option] -or-
              SYSTAT [option C] -or-
              SYSTAT [option C ##]

              -where-

              ->  option =
                  M(emory) P(rocesses) O(verview)
                  U(ser Processes) Q(ueues) H(elp)

              ->  C = Continuous display
                 ## = 1-2 digits indicating the period,
                        in seconds, between display refreshes.

Type any letter to return to the menu.
```

The M, P, Q, and U options ask you if you prefer a continuous display. If you type y, Concurrent CP/M-86 asks for a time interval, in seconds, and then displays a real-time window of information. If you type n, a static snapshot of the requested information appears. In either case, press any key to return to the menu.

The M(emory) option displays all memory potentially available to you, but it does not display restricted memory. The partitions are listed in memory-address order. Length parameter is shown in paragraph values.

The P(rocess) option displays all system processes and the resources they are using.

The Q(ueues) option displays all system queues, listing queue readers, writers, and owners.

The U(ser Processes) option displays only user-initiated processes in the same format as the P(rocess) option.

The O(verview) option displays an overview of the system parameters, as specified at system generation time. The display is not continuous.

The E(xit) option returns you to system level from the menu, as does CTRL-C.

End of Section 1

Section 2
The Concurrent CP/M-86 File System

2.1 File System Overview

The Basic Disk Operating System (BDOS) file system supports from one to sixteen logical drives. Each logical drive has two regions: a directory area and a data area. The directory area defines the files that exist on the drive and identifies the data area space that belongs to each file. The data area contains the file data defined by the directory.

The directory area consists of sixteen logically independent directories. These directories are identified by user numbers 0 through 15. During execution, a process runs with a system parameter called the user number set to a single value. The user number specifies the current active directories for all drives on the system. For example, the Concurrent CP/M-86 DIR utility displays only files within a directory selected by the current user number.

The file system automatically allocates directory and data area space when a process creates or extends a file, and returns previously allocated space to free space when a process deletes or truncates a file. 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, you need not be concerned with directory and drive organization when using the file system calls.

An eight-character filename and a three-character filetype field identify each file in a directory. Together, these fields must be unique for each file within a directory. However, files with the same filename and filetype can reside in different user directories without conflict. Processes can also assign an eight-character password to a file to protect it from unauthorized access.

All system calls that involve file operations specify the requested file by filename and filetype. For some system calls, multiple files can be specified by a technique called ambiguous reference. This technique uses question marks and asterisks as wildcard characters to give the file system a pattern to match as it searches a directory.

The file system supports two categories of system calls: file-access system calls and drive-related system calls. The file-access system calls have mnemonics beginning with F_, and the drive-related system calls have mnemonics beginning with DRV_. The next two sections introduce the file system calls.

2.1.1 File-access System Calls

Most of the file-access system calls can be divided into two groups: system calls that operate on files within a directory and system calls that operate on records within a file. However, the file-access category also includes several miscellaneous functions that either affect the execution of other file-access system calls or are commonly used with them.

System calls in the first file-access group include calls to search for one or more files, delete one or more files, rename or truncate a file, set file attributes, assign a password to a file, and compute the size of a file. Also included in this group are system calls to open a file, to create a file, and to close a file.

The second file-access group includes system calls to read or write records to a file, either sequentially or randomly, by record position. BDOS read and write system calls transfer data in 128 byte units, which is the basic record size of the file system. This group also includes system calls to lock and unlock records and thereby allow multiple processes to coordinate access to records within a commonly accessed file.

Before making read, write, lock, or unlock system calls for a file, you must first open or create the file. Creating a file has the side effect of opening the file for record access. In addition, because Concurrent CP/M-86 supports three different modes of opening files (Locked, Unlocked, and Read-Only), there can be other restrictions on system calls in this group that are related to the open mode. For example, you cannot write to a file that you have opened in Read-Only mode.

After a process has opened a file, access to the file by other processes is restricted until the file is closed. Again, the exact nature of the restrictions depends on the open mode. However, in all cases the file system does not allow a process to delete, rename, or change a file's attributes if another process has opened the file. Thus, the close system call performs two steps to terminate record access to a file. It permanently records the current status of the file in the directory and removes the open-file restrictions limiting access to the file by other processes.

The miscellaneous file-access system calls include calls to set the current user number, set the DMA address, parse an ASCII file specification and set a default password. This group also includes system calls to set the BDOS Multisector Count and the BDOS Error Mode. The BDOS Multisector count determines the number of 128-byte records to be processed by the read, write, lock, and unlock system calls. The Multisector count can range from 1 to 128; the default value is one. The BDOS Error Mode determines whether the file system intercepts certain errors or returns on all errors to the calling process.

2.1.2 Drive-related System Calls

BDOS drive-related system calls select the default drive, compute a drive's free space, interrogate drive status, and assign a directory label to a drive. A drive's directory label controls whether the file system enforces file password protection for files in the directory. It also specifies whether the file system is to perform date and time stamping of files on the drive.

This category also includes system calls 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. Logging in a drive initializes the drive for directory and file operations. The purpose of a drive reset call is to prepare for a media change on drives that support removable media. Under Concurrent CP/M-86, drive reset calls are conditional. A process cannot reset a drive if another process has a file open on the drive.

The following table summarizes the BDOS file system calls.

Table 2-1. File System Calls

Mnemonic	*Description*
DRV_ACCESS	Access Drive
DRV_ALLOCVEC	Get Drive Allocation Vector
DRV_ALLRESET	Reset All Drives
DRV_DPB	Get Disk Parameter Block Address
DRV_GET	Get Default Drive
DRV_GETLABEL	Get Directory Label
DRV_FLUSH	Flush Data Buffers
DRV_FREE	Free Drive
DRV_LOGINVEC	Return Drives Logged In Vector
DRV_RESET	Reset Drive
DRV_ROVEC	Return Drives R/O Vector
DRV_SETLABEL	Set Directory Label
DRV_SET	Set (Select) Drive
DRV_SETRO	Set Drive To Read-Only
DRV_SPACE	Get Free Space On Drive
F_ATTRIB	Set File's Attributes
F_CLOSE	Close File
F_DELETE	Delete File
F_DMASEG	Set DMA Segment
F_DMAGET	Get DMA Address
F_DMAOFF	Set DMA Offset
F_ERRMODE	Set BDOS Error Mode
F_LOCK	Lock Record In File
F_MAKE	Make A New File
F_MULTISEC	Set BDOS Multisector Count
F_OPEN	Open File
F_PARSE	Parse Filename
F_PASSWD	Set Default Password

Table 2-1. (continued)

Mnemonic	*Description*
F_RANDREC	Return Record Number For File Read-Write
F_READ	Read Record Sequentially From File
F_READRAND	Read Random Record From File
F_RENAME	Rename File
F_SIZE	Compute File Size
F_SFIRST	Directory Search First
F_SNEXT	Directory Search Next
F_TIMEDATE	Return File Time/Date Stamps Password Mode
F_TRUNCATE	Truncate File
F_UNLOCK	Unlock Record In File
F_USERNUM	Set/Get Directory User Number
F_WRITE	Write Record Sequentially Into File
F_WRITERAND	Write Random Record Into File
F_WRITEZF	Write Random Record With Zero Fill
F_WRITEXFCB	Write File's XFCB

The following sections contain information on important topics related to the file system. Read these sections carefully before attempting to use the system calls described individually in Section 6.

2.2 File Naming Conventions

Under Concurrent CP/M-86, a file specification consists of four parts: a drive specifier, the filename field, the filetype field, and the file password field. The general format for a command line file specification is shown below:

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

The drive specifier field specifies the drive where the file is located. The filename and filetype fields identify the file. The password field specifies the password if a file is password protected.

The drive, type, and password fields are optional, and the delimiters : . ; are required only when specifying their associated fields. The drive specifier can be assigned a letter from A to P, where the actual drive letters supported on a given system are determined by the XIOS implementation. When the drive letter is not specified, the current default drive is assumed.

The filename and password fields can contain one to eight nondelimiter characters. The filetype field can contain one to three nondelimiter characters. All three fields are padded with blanks, if necessary. Omitting the optional type or password fields implies a field specification of all blanks.

Under Concurrent CP/M-86, the P_CLI system call interprets ASCII command lines and loads programs. The P_CLI system call makes F_PARSE system calls to parse file specifications from a command line. F_PARSE recognizes certain ASCII characters as delimiters when it parses a file specification. These characters are shown in Table 2-2.

Table 2-2. Valid Filename Delimiters

ASCII	*Hex Equivalent*
null	000H
space	020H
return	00DH
tab	009H
:	03AH
.	02EH
;	03BH
=	03DH
,	02CH
[	05BH
]	05DH
<	03CH
>	03EH
|	07CH

The F_PARSE system call also excludes all control characters from the file specification fields and translates all lower-case letters to upper-case.

Avoid using parentheses and the backslash character, \, in the filename and filetype fields because they are commonly used delimiters. Use asterisk and question mark characters, * and ?, only to make an ambiguous file reference. When F_PARSE encounters an asterisk in a filename or filetype field, it pads the remainder of the field with question marks. For example, a filename of X*.* is parsed to X???????.???. The BDOS F_SFIRST, F_SNEXT, and F_DELETE system calls match a question mark in the filename or filetype fields to the corresponding position of any directory entry belonging to the current user number. Thus, a search operation for X???????.??? finds all the files in the current user directory beginning in X. Most other file-access BDOS system calls treat the presence of a question mark in the filename or filetype fields as an error.

It is not mandatory to follow the file naming conventions of Concurrent CP/M-86 when you create or rename a file with BDOS system calls directly from an application program. However, the conventions must be used if the file is to be accessed from a command line. For example, the P_CLI system call cannot locate a command file in the directory if its filename or filetype field contains a lower-case letter.

As a general rule, the filetype field names the generic category of a particular file, and the filename field distinguishes individual files within each category. Although they are generally arbitrary, Table 2-3 lists some of the generic filetype categories that have been established.

Table 2-3. Filetype Conventions

Filetype	*Description*
A86	8086 Assembler Source
ASM	Assembler Source
BAK	Text or Source Back-up
BAS	BASIC Source File
C	C Source File
CMD	8086 Command File
COM	8080 Command File
CON	CCP/M-86 Modules
DAT	Data File
HEX	HEX Machine Code
H86	ASM86 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	Symbol File
SYS	System File
$$$	Temporary File

2.3 Disk Drive and File Organization

The file system can support up to sixteen logical drives, identified by the letters A through P. A logical drive usually corresponds to a physical drive on the system, particularly for physical drives that support removable media such as floppy disks. High-capacity hard disks, however, are commonly divided up into multiple logical drives. If a disk contains system tracks reserved for the boot loader, these tracks precede the tracks of the disk mapped by the logical drive. In this manual, references to drives means logical drives, unless explicitly stated otherwise.

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 space to files. Table 2-4 displays the relationship between data block size and total drive capacity.

Table 2-4. Drive Capacity

Data Block Size	*Maximum Drive Capacity*
1K	256 kilobytes
2K	64 megabytes
4K	128 megabytes
8K	256 megabytes
16K	512 megabytes

Each drive is divided into two regions: a directory area and a data area. The directory area contains from one to sixteen blocks located at the beginning of the drive. The actual number is set in the XIOS. Directory entries residing in this area define the files that exist on the drive. In addition, the directory entries belonging to a file identify the data blocks in the drive's data area that 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.

Each disk file consists of a set of up to 262,144 (40000H) 128-byte records. Each record of a file is identified by its position in the file. This position is called the record's Random Record Number. If a file is created sequentially, the first record has a position of zero, while the last record has a position one less than the number of records in the file. Such a file can be read sequentially, beginning at record zero, or randomly by record position. Conversely, if a file is created randomly, records are added to the file by specified position. A file created in this way is called sparse if positions exist within the file where a record has not been written.

The BDOS automatically allocates data blocks to a file to contain the file's records on the basis of the record positions consumed. Thus, a sparse file that contains two records, one at position zero, the other at position 262,143, consumes only two data blocks in the data area. Sparse files can only be created and accessed randomly, not sequentially. Note that any data block allocated to a file is permanently allocated until the file is deleted or truncated. These are the only mechanisms supported by the BDOS 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 character (1AH), or a real end-of-file, returned by the BDOS read system call. Note that these source file conventions are not supported in the file system directly but are followed by Concurrent CP/M-86 utilities such as TYPE and ASM-86™. In addition, CTRL-Z characters embedded within other types of files such as CMD files do not signal end-of-file.

2.4 File Control Block Definition

The File Control Block (FCB) is a system data structure that serves as an important channel for information exchange between a process and BDOS file-access system calls. A process initializes an FCB to specify the drive location, filename and filetype fields, and other information that is required to make a file-access call. For example, in an F_OPEN system call, the FCB specifies the name and location of the file to be opened. In addition, the file system uses the FCB to maintain the current state and record position of an open file. Some file-access system calls use special fields within the FCB for invoking options. Other file-access system calls use the FCB to return data to the calling program. All BDOS random I/O system calls require the calling process to specify the Random Record Number in a 3-byte field at the end of the FCB.

When a process makes a BDOS file-access system call, it passes an FCB address to the BDOS. This address has two 16-bit components: register DX, which contains the offset, and register DS, which contains the segment. The length of the FCB data area depends on the BDOS system call. For most system calls, the minimum length is 33 bytes. For the F_READRAND, F_WRITERAND, F_WRITEZF, F_LOCK, F_UNLOCK, F_RANDREC, F_SIZE, and F_TRUNCATE system calls, the minimum FCB length is 36 bytes. When the F_OPEN or F_MAKE system calls open a file in Unlocked mode, the FCB must be at least 35 bytes long. Figure 2-1 displays the FCB data structure in two formats.

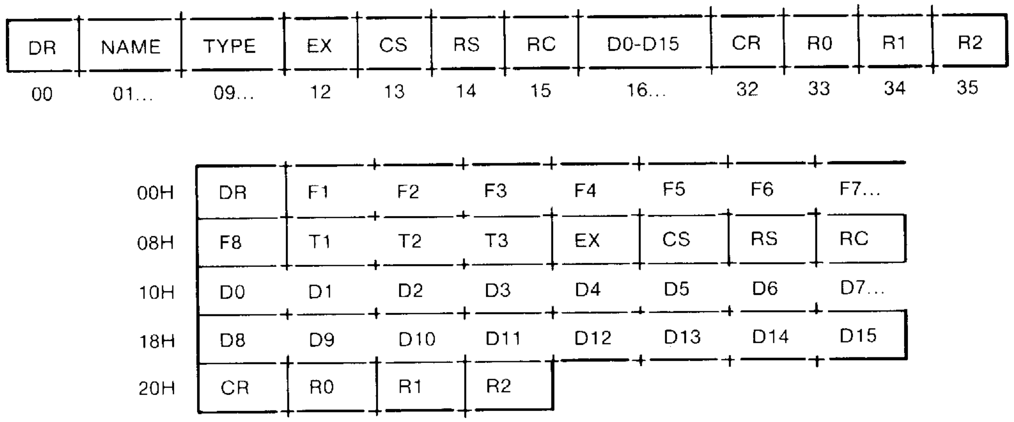
DR NAME TYPE EX CS RS RC D0-D15 CR R0 R1 R2
00 01... 09... 12 13 14 15 16... 32 33 34 35
00H DR F1 F2 F3 F4 F5 F6 F7...
08H F8 T1 T2 T3 EX CS RS RC
10H D0 D1 D2 D3 D4 D5 D6 D7...
18H D8 D9 D10 D11 D12 D13 D14 D15
20H CR R0 R1 R2

Figure 2-1. FCB - File Control Block

The fields in the FCB are defined as follows:

Table 2-5. FCB Field Definitions

Field	*Definitions*
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 called 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 also called attribute bits. T1' = 1 => Read-Only file, T2' = 1 => System file, T3' = 1 => File has been archived.
EX	Contains the current extent number. This field 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.

Table 2-5. (continued)

Field	*Definitions*
RS	Reserved for internal system use.
RC	Record count for extent EX. This field takes on values from 0 to 255 (values greater than 128 imply a record count of 128).
D0...D15	Normally filled in by Concurrent CP/M-86 and reserved for system use. Also used to specify the new filename and filetype with the F_RENAME system call.
CR	Current record to read or write in a sequential file operation. This field is normally set to zero by the calling process when a file is opened or created.
R0,R1,R2	Optional Random Record Number in the range 0-262,143 (0 - 3FFFFH). R0, 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 of the FCB when a file is successfully opened in Unlocked mode (refer to Section 2.14).

2.4.1 FCB Initialization and Usage

The calling process must initialize bytes 0 through 11 of the referenced FCB before making the following file-access system calls: F_ATTRIB, F_DELETE, F_MAKE, F_OPEN, F_RENAME, F_SFIRST, F_SIZE, F_SNEXT, F_TIMEDATE, F_TRUNCATE, and F_WRITEXFCB. Normally, the DR field specifies the drive location of the file, and the name and type fields specify the name of the file. You must also set the EX field of the FCB before calling F_MAKE, F_OPEN, F_SFIRST, and F_WRITEXFCB. Except for the F_WRITEXFCB system call, you can usually set this field to zero. Note that the F_RENAME system call requires the calling process to place the new filename and filetype in bytes D1 through D11.

The remaining file-access calls that use FCBs require an FCB that has been initialized by a prior file-access system call. For example, the F_SNEXT system call expects an FCB initialized by a prior F_SFIRST call. In addition, the F_LOCK, F_READ, F_READRAND, F_UNLOCK, F_WRITERAND, and F_WRITEZF system calls require an FCB that has been activated for record operations. Under Concurrent CP/M-86, only the F_OPEN and F_MAKE system calls can activate an FCB.

If you intend to process a file sequentially from the beginning, using the F_READ and F_WRITE system calls, you must set byte 32 to zero before you make your first read or write call. In addition, when you make a F_LOCK, F_READRAND, F_UNLOCK, F_WRITERAND, or F_WRITEZF system call, you must set bytes R0 through R2 of the FCB to the requested Random Record Number. The F_TRUNCATE system calls also requires the FCB random record field to be initialized.

The F_SFIRST, F_SNEXT, and F_DELETE system calls support multiple or ambiguous reference. In general, a question mark in the filename, filetype, or EX fields matches all values in the corresponding positions of directory entries during a directory search operation. File directory entries maintained in the directory area of each disk drive have the same format as FCBs except for byte 0, which contains the file's user number, and bytes 32 through 35, which are not present. The search system calls, F_SFIRST and F_SNEXT, also recognize a question mark in the FCB DR field, and, if specified, they return all directory entries on the disk regardless of user number, including empty entries. A directory FCB that begins with E5H is an empty directory entry.

When the F_OPEN and F_MAKE system calls activate an FCB for record operations, they copy the FCB's matching directory entry from disk, excluding byte 0, into the FCB in memory. In addition, these system calls compute and store a checksum value in the CS field of the FCB. During subsequent record operations on the file, the file system uses this checksum field to verify that the FCB has not been modified by the calling process in an illegal way. Thus, all read, write, lock, and unlock operations on a file must specify a valid activated FCB; otherwise, the BDOS returns a checksum error. The BDOS performs this checking to protect the integrity of the file system. In general, you should not modify bytes 0 through 31 of an open FCB, except to set interface attributes (see Section 2.4.3). Other restrictions related to activated FCBs are discussed in Section 2.10.

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 also updates the memory copy of the FCB to record the allocation of data blocks to the file. At the termination of file processing, the F_CLOSE system call permanently records this information on disk.

Note that the BDOS does not record the data blocks allocated to a file during write operations in the disk directory until the calling process issues an F_CLOSE call. Therefore, a process that creates or modifies files must close the files at the termination of file processing. Otherwise, data might be lost.

2.4.2 File Attributes

The high-order bits of the FCB filename (F1',...,F8') and filetype fields (T1',T2',T3') are called attribute bits. Attribute bits are 1-bit Boolean fields, where 1 indicates on or true, and 0 indicates off or false. Attribute bits indicate two kinds of attributes within the file system: file attributes and interface attributes. The file attributes are described in this section. Section 2.4.3 describes interface attributes.

The file attribute bits, 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 F_ATTRIB system call. When the F_MAKE system call creates a file, it initializes all file attributes to zero. A process can interrogate file attributes in an FCB activated by the F_OPEN system call, or in directory FCBs returned by the F_SFIRST and F_SNEXT system calls.

Note: the file system ignores the file attribute bits when it attempts to locate a file in the directory.

The file system defines file attributes T1',T2',and T3' as follows:

Table 2-6. File Attribute Definitions

Attribute	*Definition*
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. The Concurrent CP/M-86 DIR utility does not usually display System files. In addition, user-zero system files can be accessed on a Read-Only basis from other user numbers.
T3': Archive Attribute	User-written archive programs use this attribute. When an archive program copies a file to back-up storage, it sets the archive attribute of the copied files. The file system automatically resets the archive attribute of a directory entry when writing to the directory entry's region of a file. An archive program can test this attribute in each of the file's directory entries using the F_SFIRST and F_SNEXT system calls. If all directory entries have the archive attribute set, the file has not been modified since the previous archive. The Concurrent CP/M-86 PIP utility supports file archival.

File attributes F1' through F4' of command files are defined as Compatibility Attributes under Concurrent CP/M-86 (see Section 2.12). However, for all other files, attributes F1' through F4' are available for definition by the user.

2.4.3 Interface Attributes

The interface attributes are F5', F6', F7', and F8'. These attributes cannot be used as file attributes. Interface attributes F5' and F6' request options for BDOS file-access system calls. Table 2-7 lists the F5' and F6' attribute definitions for the system calls that define interface attributes. Note that the F5' = 0 and F6' = 0 definitions are not listed if their definition simply implies the absence of the option associated with setting the interface attribute.

Table 2-7. BDOS Interface Attributes F5' and F6'

System Call	*Attribute*
F_ATTRIB	F5' = 1 : Maintain extended file lock
	F6' = 1 : Set file byte count
F_CLOSE	F5' = 1 : Partial Close
	F6' = 1 : Extend file lock
F_DELETE	F5' = 1 : Delete file XFCBs only and maintain extended file lock
F_LOCK	F5' = 0 : Exclusive Lock
	F5' = 1 : Shared Lock
	F6' = 0 : Lock existing records only
	F6' = 1 : Lock logical records
F_MAKE	F5' = 0 : Open in Locked mode
	F5' = 1 : Open in Unlocked mode
	F6' = 1 : Assign password to file
F_OPEN	F5' = 0 : Open in Locked mode
	F5' = 1 : Open in Unlocked mode
	F6' = 0 : Open in mode specified by F5'
	F6' = 1 : Open in Read-Only mode
F_RENAME	F5' = 1 : Maintain extended file lock
F_TRUNCATE	F5' = 1 : Maintain extended file lock
F_UNLOCK	F5' = 1 : Unlock all locked records

Section 6 details the above interface attribute definitions for each of the preceding system calls. Note that the BDOS always resets interface attributes F5' and F6' before returning to the calling process. Interface attributes F7' and F8' are reserved for internal use by the file system.

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 extend across more than one user number.

Only one user number is active for a specific process at one time. For this process, the current user number applies to all drives on the system. Furthermore, the FCB format does not contain a field that can override the current user number. As a result, all file and directory operations reference only directory entries associated with the current user number.

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 F_USERNUM system call before issuing the BDOS call. However, 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 file system returns an FCB checksum error.

When the P_CLI system call initiates a transient process or Resident System Process (described in detail in Section 5), it sets the user number to the default value established by the process issuing the P_CLI system call. The sending process is usually the TMP. However, the sending process can be another process, such as a transient program that makes a P_CHAIN call. A transient process can change its user number by making a F_USERNUM call. Changing the user number in this way does not affect the command line user number displayed by the TMP. Thus, when a transient process that has changed its user number terminates, the TMP restores and displays the original user number in the command line prompt when it regains control.

User 0 has special properties under Concurrent CP/M-86. The file system automatically opens files listed under user zero but requested under another user number if the file is not present under the current user number, and if the file on user zero has the system attribute (T2') set. This convention allows utilities, including overlays and any other commonly accessed files, to reside on user zero, but remain available to other users. This eliminates the need to copy commonly used utilities to all user numbers on a directory, and gives the Concurrent CP/M-86 user control over which files are accessible to the different user areas.

2.6 Directory Labels and XFCBs

The file system includes three special types of FCBs: the directory label and the XFCB, described in this section, and the SFCB, described in detail in Section 2.8.

The directory label specifies for its drive whether password support is to be activated, and if date and time stamping for files is to be performed. The format of the directory label is shown below in Figure 2-2.

DR	NAME	TYPE	DL	S1	S2	RC	PASSWORD	TS1	TS2
00	01...	09...	12	13	14	15	16...	25.	29.

Figure 2-2. Directory Label Format

Table 2-8. Directory Label Field Definitions

Field	*Definition*
DR	drive code (0 - 16)
Name	directory label name
Type	directory label type
DL	directory label data byte Bit 7 - enable password support Bit 6 - perform access time stamping Bit 5 - perform update time stamping Bit 4 - perform create time stamping Bit 0 - Directory Label exists (Bit references are right to left, relative to 0)
S1,S2,RC	reserved for future use
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 area. The directory label name and type fields are not used to search for a directory label; they can be used to identify a disk.

You can use the DRV_SETLABEL system call to create a directory label or update its fields. This system call can also assign a password to a directory label. The directory label password, if assigned, cannot be circumvented, whereas file password protection on a drive is an option controlled by the directory label. Thus, access to the directory label password provides the ability to bypass password protection on the drive.

Note: the file system provides no specific system call to read the directory label FCB directly. However, you can read the directory label data byte directly with the BDOS system call, DRV_GETLABEL. In addition, you can use the BDOS search system calls F_SFIRST and F_SNEXT to find a directory label. You can identify the directory label by a value of 32 (020H) in byte 0 of the directory FCB.

The XFCB is an extended FCB that can optionally be associated with a file in the directory. If present, it contains the file's password and password mode. The format of the XFCB is shown below in Figure 2-3.

DR	FILE	TYPE	PM	S1	S2	RC	PASSWORD	RESERVED	
00	01...	09...	12	13	14	15	16......	25.	29.

Figure 2-3. XFCB - Extended File Control Block

The fields in the XFCB are defined in Table 2-9:

Table 2-9. XFCB Field Definitions

Field	*Definition*
DR	drive code (0 - 16)
File	filename field
Type	filetype field
PM	password mode Bit 7 - Read mode Bit 6 - Write mode Bit 5 - Delete mode (Bit references are right to left, relative to 0)
S1,S2,RC	reserved for system use
Password	8-byte password field (encrypted)
Reserved	8-byte area reserved for future use

An XFCB can only be created on a drive that has a directory label, and only if the directory label enables password protection. For drives in this state, there are two ways to create an XFCB for a file: with the F_MAKE system call or the F_WRITEXFCB system call. The F_MAKE system call creates an XFCB if the calling process requests that a password be assigned to the created file. The F_WRITEXFCB system call creates an XFCB when it is called to assign a password to an existing file. You can identify an XFCB in the directory by a value of 16 (010H) + N in byte 0 of the FCB, where N equals the user number.

2.7 File Passwords

There are two ways to assign passwords to a file: by the F_MAKE system call or by the F_WRITEXFCB system call. You can also change a file's password or password mode with the F_WRITEXFCB system call if you can supply the original password. Note that you cannot change a file's password or password mode if password protection for the drive is disabled by the directory label. However, even if you cannot supply a file's password, you can delete a file's XFCB, thereby removing its password protection, if password protection is disabled on the drive.

The Concurrent CP/M-86 BDOS provides password protection in one of three modes when password support is enabled by the directory label. Table 2-10 shows the difference in access level allowed to BDOS system calls when the password is not supplied.

Table 2-10. Password Protection Modes

Mode	*Access Level Allowed Without Password*
(1) Read	Cannot be read, modified, or deleted.
(2) Write	Can be read, but not modified or deleted.
(3) Delete	Can be read and modified, but not deleted.

If a file is password protected in Read mode, a process must supply the password to open the file. Processes cannot write to a file protected in Write mode without the password. A file protected in Delete mode allows read and write access, but a process must specify the password to delete or truncate 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 or truncate the file, rename the file, or to modify the file's attributes.

If a process supplies the correct password or the directory label disables password protection, then access to the BDOS system calls is the same as for a file that is not password-protected. In addition, the F_SFIRST and F_SNEXT system calls are not affected by file passwords. The following BDOS system calls test for passwords.

DRV_SETLABEL
F_ATTRIB
F_DELETE
F_OPEN
F_RENAME
F_WRITEXFCB
F_TRUNCATE

The BDOS maintains file passwords in the XFCB and directory label in encrypted form. To make a BDOS system call for a file that requires a password, a process must place the password in the first eight bytes of the current DMA, or make it the default password with the F_PASSWD system call, before making the system call.

Note: the BDOS maintains the assigned default password for each process. Processes inherit the default password of their parent process. You can set a given TMP's default password using the SET command; all programs loaded by this TMP inherit the same default password.

2.8 File Date and Time Stamps: SFCBs

The Concurrent CP/M-86 file system uses a special type of directory entry called an SFCB to record date and time stamps for files. When a directory has been initialized for date and time stamping, SFCBs reside in every fourth position of the directory. Each SFCB maintains the date and time stamps for the previous three directory entries, as shown in Figure 2-4.

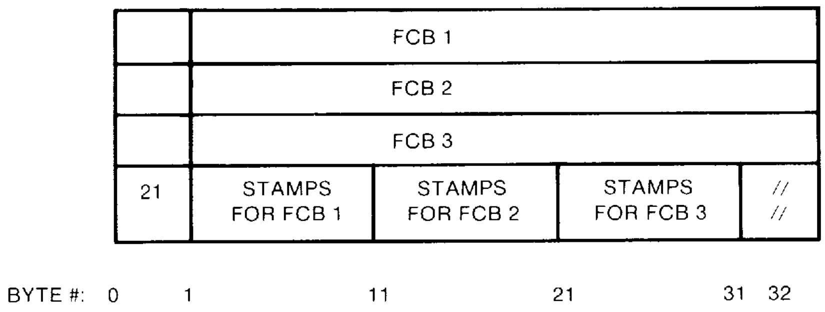

Figure 2-4. Directory Record with SFCB

This figure shows a 128-byte directory record containing an SFCB. Directory records have four directory entries, each 32 bytes long; SFCBs always occupy the last 32-byte entry in the directory record.

The SFCB itself contains five fields. The first field is a single byte containing the value 021H; this field identifies the SFCB within the directory. The next three fields, called the SFCB subfields, are each 10 bytes in length and contain the date and time stamps for their corresponding FCB entries in the directory record. The last byte of the SFCB is reserved for system use. Figure 2-5 shows the detail of the SFCB subfields.

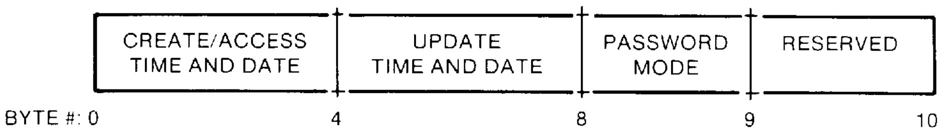

Figure 2-5. SFCB Subfields

An SFCB subfield only contains valid information if its corresponding FCB in the directory record is an extent zero FCB. This FCB is a file's first directory entry. For password protected files, the SFCB subfield also contains the password mode of the file; the password mode field is zero for files without password protection. You can read SFCBs by making F_SFIRST and F_SNEXT system calls. In addition, you can make a F_TIMEDATE system call to retrieve the date and time stamps and password mode of a specified file. Refer to the T_GET system call definition in Section 6 for the description of the format of a date and time stamp field.

Concurrent CP/M-86 supports three kinds of file stamping: create, access, and update. Create stamps record when the file was created, access stamps record when the file was last opened, and update stamps record the last time the file was modified. Create and access stamps share the same field. As a result, file access stamps overwrite any create stamps.

The directory label of a properly initialized disk determines the type of date and time stamping for files on the drive. The INITDIR utility initializes a directory for date and time stamping by placing an SFCB in every fourth directory entry. Disks not initialized in this way cannot support date and time stamping. In addition, date and time stamping is not performed if the disk's directory label is absent or does not specify date and time stamping, or if the disk is Read-Only.

Note that the directory label is also time stamped, but these stamps are not made in an SFCB; time stamp fields in the last eight bytes of the directory label show when it was created and last updated. Access stamping is not supported for directory labels.

The BDOS file system uses the system date and time when it records a date and time stamp. This value is maintained in a field in the SYSDAT part of the System Data Segment. The DATE utility sets the system time and date (refer to the *Concurrent CP/M-86 User's Guide* for details of using DATE).

2.9 File Open Modes

The 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 process issuing the F_OPEN call cannot supply the password. In both of these cases, the BDOS allows only read operations to the file.

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 another process has already opened the file 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 process issuing the F_OPEN call cannot supply the password.

When opening a file in Unlocked mode, a process must reserve 35 bytes in the FCB because the F_OPEN system call 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 F_LOCK and F_UNLOCK system calls. These BDOS system calls work only for files opened in Unlocked mode.

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 another process has opened the file in Read-Only mode. This mode allows more than one process to open the same file for Read-Only access.

The F_OPEN system call performs the following steps for files opened in Locked or Read-Only mode. If the current user number is nonzero, and the file to be opened does not exist under the current user number, the F_OPEN system call searches the user zero directory for the file. If the file exists under user zero and has the system attribute T2' set, the BDOS opens the file under user zero. The open mode is automatically forced to Read-Only when this is done.

The F_OPEN system call 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. The Read-Only attribute controls whether a user-zero process and processes on other user numbers can concurrently open a user-zero system file 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 the BDOS allows concurrent access of the file. However, if the Read-Only attribute is reset, the user-zero process opens the file in Locked mode. This prevents sharing the file with other processes.

The F_OPEN and F_MAKE system calls use FCB interface attributes F5' and F6' to specify the open mode. The interface attribute definitions for these functions are listed in Table 2-7.

Note: the F_MAKE system call does not allow opening the file in Read-Only mode.

2.10 File Security

In general, the security measures implemented in the file system prevent accidental collisions between running processes. It is not possible to provide total security under Concurrent CP/M-86 because the file system maintains file allocation information in open FCBs in the user's memory region, and Concurrent CP/M-86 does not require memory protection. However, the 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.

The BDOS validates the checksum of user FCBs before all I/O operations to protect the integrity of the file system from corrupted FCBs. The F_OPEN and F_MAKE system calls compute and assign checksums to FCBs. The F_READRAND, F_READ, F_WRITERAND, F_WRITEZF, F_WRITE, F_LOCK, and F_UNLOCK system calls subsequently verify and recompute the checksums when they change the FCB. The F_CLOSE system call also verifies FCB checksums. Note that FCB verification by these system calls can be disabled (see Section 2.12), but Concurrent CP/M-86's file security is reduced when this is done. If the BDOS detects an FCB checksum error, it does not perform the requested command. Instead, it either returns to the calling process with an error code, or if the system call is F_CLOSE and the BDOS Error mode is in the default state (see Section 2.18), it terminates the calling process with an error message.

Concurrent CP/M-86 uses a system data structure, called the Lock List, to manage file opening and record locking by running processes. Each time a process opens a file or locks a record successfully, the file system allocates an entry in the system Lock List to record the fact. The file system uses the following information to

- prevent a process from deleting, truncating, 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 unlocked or Read-Only mode.
- prevent a process from resetting a drive on which another process has an open file.
- prevent a process from reading, writing, or locking a record currently locked by another process. Refer to Section 2.14 for more information on record locking and unlocking.

The file system only verifies whether another process has the FCB-specified file open for the following file-access system calls: F_OPEN, F_MAKE, F_DELETE, F_RENAME, F_ATTRIB, and F_TRUNCATE. For file-access system calls that require an open FCB, the FCB checksum controls whether the calling 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.

The most common way a process releases a lock entry for an open file is by closing the file. A close operation is permanent if it causes the removal of the file's open lock list entry. The file system invalidates the FCB checksum field on permanent close operations to prevent continued open file operations with the FCB.

However, not all close operations are permanent. For example, if a process makes multiple F_OPEN or F_MAKE calls to an open file, a matching number of F_CLOSE calls must be made before the file system permanently closes the file. Of course, if you only open a file once, a single close operation permanently closes the file. In addition, a process can optionally make partial F_CLOSE calls to a file by setting interface attribute F5'. A partial close operation does not affect the open state of a file. In the above example, a partial close operation would not count against an F_OPEN or F_MAKE call. A partial close operation simply updates the directory to reflect the current state of the file.

As a general rule, under Concurrent CP/M-86 a process should close files as soon as it no longer needs them, even if it has not modified them. While a process has a file open, access by other processes to the file is restricted. For example, after a process has opened a file in Locked mode, the file cannot be opened by other processes until the file is closed or the process terminates.

Furthermore, space in the system Lock List is limited. If a process attempts to open a file and no space remains in the system Lock List, or if the process exceeds the open file limit, the BDOS denies the open request and usually terminates the calling process. You can change the way the file system handles this error by making a F_ERRMODE system call. Note that the size of the system Lock List and the process open file limit are GENCCPM parameters.

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 F_DELETE call for a file it has open in Locked mode, the file system deletes the file and also purges the file's entry from the system Lock List. Deleting an open file is not recommended under Concurrent CP/M-86 but it is supported for files opened in Locked mode to provide compatibility with software written under earlier releases of MP/M™ and CP/M. The file system does not allow deletion of a file opened in Unlocked or Read-Only mode.

To ensure that the process does not use the open FCB corresponding to the deleted file, the file system subsequently checks all open FCBs for the process. Each open FCB is checked the next time it is used with a file-access system call that requires an open FCB. If a Lock List entry exists for the file, the BDOS allows the operation to proceed; if not, it indicates that the file has been purged and the file system returns an FCB checksum error.

The file system performs this verification of a process's open FCBs whenever it purges an open file entry from the system Lock List. The following list describes these situations:

- A process makes a F_ATTRIB, F_DELETE, F_RENAME, or F_TRUNCATE system call to a file it has open in Locked mode. These operations cannot be performed on a file open in Unlocked or Read-Only mode.

- A process issues a DRV_FREE call for a drive on which it has an open file.
- The BDOS detects a change in media on a drive that has open files. This is a special case because a process cannot control the occurrence of this situation, and because it can impact more than one process. Refer to Section 2.17 for more details on this situation.

Open FCB verification can affect performance because each verification operation requires a directory search operation. In general, you should avoid such situations when creating new programs for Concurrent CP/M-86.

2.11 Extended File Locking

Extended file locking enables a Concurrent CP/M-86 process to maintain a lock on a file after the file is permanently closed. This facility allows a process to set the attributes, delete, rename, or truncate a file without interference from other processes. In addition, this technique avoids the problems associated with using these system calls on open files (see Section 2.10).

A process can also reopen a file with an extended lock and continue open file processing. To illustrate how extended file locking might be used, a process can close an open file, rename the file, reopen the file under its new name, and continue with file operations without ever losing the file's Lock List item and control over the file.

A process can only specify extended file locking for a file it has opened in Locked mode. To extend a file's lock, set interface attribute F6' when closing the file. The F_CLOSE system call interrogates this attribute only when it is closing a file permanently. Thus, interface attribute F5', signifying a partial close, must be reset when the F_CLOSE call is made. In addition, the close operation must be permanent. If a process has opened a file N times, the F_CLOSE system call ignores the F6' attribute until the file is closed for the Nth time.

Note that the access rules for a file with an extended lock are identical to the rules for a file open in Locked mode.

To maintain an extended file lock through a F_ATTRIB, F_RENAME, or F_TRUNCATE system call, set interface attribute F5' of the referenced FCB when making the call. The BDOS honors this attribute only if the file has been closed with an extended lock. Setting attribute F5' also maintains an extended file lock for the F_DELETE system call, but setting this attribute also changes the nature of the delete operation to an XFCB-only delete. If successful, all four of these system calls delete a file's extended lock item if they are called with attribute F5' reset. However, the extended lock item is not deleted if they return with an error code.

You can make a F_OPEN call to resume record operations on a file with an extended lock. Note that you can also change the open mode when you reopen the file. The following example illustrates the use of extended locks.

1. Open file EXLOCK.TST in Locked mode.
2. Perform read and write operations on the file EXLOCK.TST using the open FCB.
3. Close file EXLOCK.TST with interface attribute F6' set to retain the file's lock item.
4. Use the F_RENAME system call to change the name of the file to EXLOCK.NEW with interface attribute F5' set to retain the file's extended lock item.
5. Reopen the file EXLOCK.NEW in Locked mode.
6. Perform read and write operations on the file EXLOCK.NEW, using the open FCB.
7. Close file EXLOCK.NEW again with interface attribute F6' set to retain the file's lock item.
8. Set the Read-Only attribute and release the file's lock item by making a F_ATTRIB system call with interface attribute F5' reset.

At this point, the file EXLOCK.NEW becomes available for access by another process.

2.12 Compatibility Attributes

Compatibility attributes provide a mechanism to modify some of the Concurrent CP/M-86 file security rules for specific command files. Concurrent CP/M-86 includes this facility because some programs developed under earlier Digital Research operating systems do not run properly under Concurrent CP/M-86. Most of the problems encountered by these programs occur because they were designed for single-tasking operating systems where file security is not required. For example, a program might close a file and then continue reading and writing to the file. Under CP/M-86, this does not cause a problem. However, under Concurrent CP/M-86, the file system intercepts open file operations with a deactivated FCB to ensure the integrity of the file system. With compatibility attributes, you have a tool for dealing with these kinds of situations.

You should only use compatibility attributes with existing programs that run properly under CP/M or CP/M-86. Do not use compatibility attributes with new programs you develop under Concurrent CP/M-86.

Compatibility attributes are defined as file attributes F1' through F4' of program (CMD) files. You can use the Concurrent CP/M-86 SET utility to set these file attributes from the command line. However, setting a command file's compatibility attributes has no affect unless the GENCCPM COMPATMODE option has been selected during system generation. If this has been done, the P_CLI system call interrogates file attributes F1' through F4' of the command file during program loading and modifies the Concurrent CP/M-86 file security rules for the loaded program.

The Concurrent CP/M-86 BDOS defines the compatibility attributes as shown in Table 2-11.

Table 2-11. Compatibility Attribute Definitions

Attribute	*Definition*
F1' Modify the rules for Locked mode.	
	When a process running with F1' set opens a file in Locked mode, it can perform read and write operations to the file as normal. However, to other processes on the system, it appears as if the file was opened in Read-Only mode. Thus, another process running with F1' set, can open the same file in Locked mode and also perform write operations to the file. In addition, if a process with F1' reset attempts to open the file in Locked or Read-Only mode, the open attempt is allowed but the open mode is forced to Read-Only. Furthermore, write operations are not allowed when the process has F1' reset.
	This compatibility mode is designed to allow multiple copies of the same program to run concurrently, even though the program might make read and write calls to a common file that it has opened in Locked mode. In addition, this compatibility mode allows other programs not in this compatibility mode to access the file on a Read-Only basis. Note that record locking is not supported for this modified open mode. In addition, to be safe, make all static files such as program and help files Read-Only if you use this compatibility attribute.
	There is an alternative to using this attribute if a program only makes read calls to the common file. By placing the file under User 0 with the SYS and Read-Only attributes set, you force the open mode to Read-Only when the file is opened in Locked mode.

Table 2-11. (continued)

Attribute	*Definition*
F2' Change F_CLOSE to partial close.	
	Processes running with F2' set, only make partial F_CLOSE system calls. This attribute is intended for programs that close a file to update the directory but continue to use the file. A side effect of this attribute is that files opened by a process are not released from the system Lock List until the process terminates. When using this attribute, it might be necessary to set the system Lock List parameters to higher values when you generate a system with GENCCPM.
F3' Ignore close checksum errors.	
	This attribute changes the way the F_CLOSE system call handles Close Checksum errors. Normally, the file system prints an error message on the console and terminates the calling process. However, if this attribute is set, the F_CLOSE system call ignores the checksum error and performs the close operation. This interface attribute is intended for programs that modify an open FCB before closing a file.
F4' Disable FCB Checksum verification for read and write operations.	
	Setting this attribute also sets attributes F2' and F3'. This attribute is intended for programs that modify open FCBs during read and write operations. Use this attribute very carefully, and only with software known to work, because it effectively disables Concurrent CP/M-86's file security.

Use the Concurrent CP/M-86 SET utility to specify the combination of compatibility attributes you want set in the program's command file. For example,

```
A>SET filespec [f1=on]
A>SET filespec [f1=on,f3=on]
A>SET filespec [f4=on]
```

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

- If the program ends with the "File Currently Opened" message 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 Read-Only 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.

2.13 Multisector I/O

The BDOS file system provides the capability to read or write multiple 128-byte records in a single BDOS system 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. In addition, the BDOS file system bypasses, when possible, all intermediate record buffering during multisector I/O operations. Data is transferred directly between the calling process's memory and the drive. The BDOS also informs the XIOS when it is reading or writing multiple physical records on a drive. The XIOS can use this information to further optimize the I/O operation resulting in even better performance. As a result, the use of this facility in an application program can improve its performance and also enhance overall system throughput, particularly when performing sequential I/O.

The number of records that can be transferred with multisector I/O ranges from 1 to 128. This value, called the BDOS Multisector Count, can be set by the F_MULTISEC system call. The P_CLI system call sets the Multisector Count to one when it initiates a transient program for execution. Note that the greatest potential performance increases are obtained when the Multisector Count is set to 128. Of course, this requires a 16K buffer. The Concurrent CP/M-86 PIP utility performs its sequential I/O with a Multisector Count of 128.

The Multisector Count determines the number of operations to be performed by the following BDOS system calls:

- F_READ and F_WRITE system calls
- F_READRAND, F_WRITERAND, and F_WRITEZF
- F_LOCK and F_UNLOCK

If the Multisector Count is N, calling one of the above system calls is equivalent to making N system calls. With the exception of disk I/O errors encountered by the XIOS, if an error interrupts a multisector read or write operation, the file system returns the number of 128-byte records successfully transferred in register AH. Section 2.14 describes how the Multisector Count affects the F_LOCK and F_UNLOCK system calls.

2.14 Concurrent File Access

Concurrent CP/M-86 supports two open modes, Read-Only and Unlocked, which allow concurrently running processes to access common files for record operations. The Read-Only open mode allows multiple processes to read from a common file, but processes cannot write to a file open in this mode. Thus, files remain static when they are opened in Read-Only mode. The Unlocked open mode is more complex because it allows multiple processes to read and write records to a common file. As a result, Unlocked mode has some important differences 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 Concurrent CP/M-86's record locking system calls, F_LOCK and F_UNLOCK, which are only supported for files open in Unlocked mode. Note that these system calls return a successful error code if they are called for files opened in Locked mode. However, they perform no action in this case, because, by definition, the calling process has the entire file locked.

The F_LOCK and F_UNLOCK system calls allow a process to establish and release temporary ownership to particular records within a file. You must set the FCB Random Record field and place the File ID in the first two bytes of the current DMA buffer before making these calls. The file system locks and unlocks records in units of 128 bytes, which is the standard Concurrent CP/M-86 record size. The number of records locked or unlocked is controlled by the BDOS Multisector count, which can range from 1 to 128 (see Section 2.13). In order to simplify the discussion of record locking and unlocking, the following paragraphs assume the Multisector count is one. However, as discussed later in this section, the more general case of multiple record locking and unlocking is a simple extension of the single record case.

The F_LOCK system call supports two types of lock operations: exclusive locks and shared locks. Interface attribute F5' specifies the type of lock. F5' = 0 requests an exclusive lock; F5' = 1 requests a shared lock. If a process locks a record with an exclusive lock, other processes cannot read, write, or lock the record. The locking process, however, can access the record with no restrictions. You should use this type of lock when exclusive control over a record is required.

If a process locks a record with a shared lock, other processes cannot write to the record or make an exclusive lock of the record. However, other processes are allowed to read the record and make their own shared locks on the record. No process, including the locking process, can write to a record with a shared lock. Shared locks are useful when you want to ensure that a record does not change, but you want to allow other processes to read the record.

The F_LOCK system call also lets you change the lock of a record if there is no conflict. For example, you can convert an exclusive lock into a shared lock with no restrictions. On the other hand, a process cannot convert a record's shared lock to an exclusive lock if another process has a shared lock on the record.

The F_LOCK system call has another option, specified by interface attribute F6', which controls whether a record must exist in order to be locked. If you make a F_LOCK system call with F6' = 0, the file system returns an error code if the specified record does not exist within the file. Setting F6' to 1 requests a logical lock operation. Logical lock operations are only limited by the maximum Concurrent CP/M-86 file size of 32 megabytes, which corresponds to a maximum Random Record Number of 262,143. You can use logical locks to control extending a shared file.

The F_UNLOCK system call is similar to the F_LOCK call except that it removes locks instead of creating them. There are few restrictions on unlock operations. Of course a process can only remove locks that it has made. The F_UNLOCK system call has one option, controlled by interface attribute F5'. If F5' is set to one, the F_UNLOCK system call removes all locks for the file made by the calling process. Otherwise, it removes the locks specified by the Random Record field and the BDOS Multisector Count. Note that the F_CLOSE system call also removes all locks for a file on permanent close operations.

If the BDOS Multisector Count is greater than one, the F_LOCK and F_UNLOCK system calls perform multiple record locking or unlocking. In general, multiple record locking and unlocking can be viewed as a sequence of N independent operations, where N equals the Multisector Count. However, if an an error occurs on any record within the sequence, no locking or unlocking is performed. For example, both F_LOCK and F_UNLOCK perform no action and return an error code if the sum of the FCB Random Record Number and the BDOS Multisector Count is greater that 262,144. As another example, the F_LOCK system call also returns an error code if another process has an exclusive lock on any record within the sequence.

When a process makes a F_LOCK system call, the file system allocates a new entry in the system Lock List to record the lock operation and associate it with the calling process. A corresponding F_UNLOCK system call removes the locked entry from the list. While the lock entry exists in the system Lock List, the file system enforces the restrictions implied by the lock item.

Because each lock item includes a record count field, a multiple lock operation normally results in the creation of a single new entry. However, if the file system must split an existing lock entry to satisfy the lock operation, an additional entry is required. Similarly, an unlock operation can require the creation of a new entry if a split is needed. Thus, in the worst case, a lock operation can require two new lock entries and an unlock operation can require one. Note that lock item splitting can be avoided by locking and unlocking records in consistent units.

These considerations are important because the Lock List is a limited resource under Concurrent CP/M-86. The file system performs no action and returns an error code if insufficient available entries exist in the system Lock List to satisfy the lock or unlock request. In addition, the number of lock items a single process is allowed to consume is a GENCCPM parameter. The file system also returns an error code if this limit is exceeded.

The file system performs several special operations for read and write system calls to a file open in Unlocked mode. These operations are required because the file system maintains the current state of an open file in the calling process'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 Locked mode.

2.15 File Byte Counts

Although the logical record size of Concurrent CP/M-86 is restricted to 128 bytes, the file system does provide a mechanism to store and retrieve a byte count for a file. This facility can identify the last byte of the last record of a file. The BDOS Compute File Size function returns the last Random Record Number, + 1, of the last record of a file.

The F_ATTRIB system call can set a file's byte count. This is an option controlled by interface attribute F6'. Conversely, the F_OPEN system call can return a file's byte count to the CR field of the FCB. The F_SFIRST and F_SNEXT system calls also return a file's byte count. These system calls return the byte count in the CS field of the FCB returned in the current DMA buffer.

Note that the file system does not access or update the byte count value in BDOS read or write system calls. However, the F_MAKE system call does set the byte count value to zero when it creates a file in the directory.

2.16 Record Blocking and Deblocking

Under Concurrent CP/M-86, the logical record size for disk I/O is 128 bytes. This is the basic unit of data transfer between the operating system and running processes. However, on disk, the record size is not restricted to 128 bytes. These records, called physical records, can range from 128 bytes to 4K bytes in size. Record blocking and deblocking is required on systems that support drives with physical record sizes larger than 128 bytes.

The process of building up physical records from 128-byte logical records is called record blocking. This process is required in write operations. The reverse process of breaking up physical records into their component 128-byte logical records is called record deblocking. This process is required in read operations. Under Concurrent CP/M-86, record blocking and deblocking is normally performed by the BDOS.

Record deblocking implies a read-ahead operation. For example, if a process reads a logical record that resides at the beginning of a physical record, the entire physical record is read into an internal buffer. Subsequent BDOS read calls for the remaining logical records access the buffer instead of the disk. Conversely, record blocking results in the postponement of physical write operations but only for data write operations. For example, if a transient program makes a BDOS write call, the logical record is placed in a buffer equal in size to the physical record size. The write operation on the physical record buffer is postponed until the buffer is needed in another I/O operation. Note that under Concurrent CP/M-86, directory write operations are never postponed.

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 the file on disk parallels the state of the file in memory after an update operation. This is only an issue on drives where physical write operations are postponed because of record blocking and deblocking. If the system should crash while a physical buffer is pending, data would be lost. To prevent this loss of data, the F_FLUSH system call can be called to force the write of any pending physical buffers associated with the calling process.

Note: the file system discards all pending physical data buffers when a process terminates. However, the file system automatically makes a F_FLUSH call in the F_CLOSE system call. Thus, it is sufficient to make a F_CLOSE system call to ensure that all pending physical buffers for that file are written to the disk.

2.17 Reset, Access, and Free Drive

The BDOS system calls DRV_ALLRESET, DRV_RESET, DRV_ACCESS, and DRV_FREE allow a process to control when to reinitialize a drive directory for file operations. This process of initializing a drive's directory is called logging-in the drive.

When you start Concurrent CP/M-86, all drives are initialized to the reset state. Subsequently, as processes reference drives, the file system automatically logs them in. Once logged-in, a drive remains in the logged-in state until it is reset by the DRV_ALLRESET or DRV_RESET system calls or a media change is detected on the drive. If the drive is reset, the file system automatically logs in the drive again the next time a process references it. The file system logs in a drive immediately when it detects a media change on the drive.

Note that the DRV_ALLRESET and DRV_RESET system calls have similar effects except that the DRV_ALLRESET system call affects all drives on the system. You can specify the combination of drives to reset with the DRV_RESET system call.

Logging-in a drive consists of several steps. The most important step is the initialization of the drive's allocation vector. The allocation vector records the allocation and deallocation of data blocks to files, as files are created, extended, deleted and truncated. Another function performed during drive log-in is the initialization of the directory checksum vector. The file system uses the checksum vector to detect media changes on a drive. Note that permanent drives, which do not support media changes, might not have checksum vectors.

Under Concurrent CP/M-86, the DRV_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 DRV_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. A drive's Removable or Nonremovable 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. If there are open files, the action taken by the reset operation depends on whether the drive is removable and whether the drive is Read-Only or Read-Write. Note that only the DRV_SETRO system call can set a drive to Read-Only. Following log-in, a drive is always Read-Write.

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 set to Read-Only, the file system determines whether other processes have open files on the drive. If they do, then it denies DRV_RESET operation and returns an error code to the calling process.

If all the open files on a removable drive belong to the calling process, the process is said to own the drive. In this case, the file system performs a qualified reset on the drive and returns a successful result. This means that the next time a process accesses this drive, the BDOS performs the log-in operation only if it detects a media change on the drive. The logic flow of the drive reset operation is shown in Figure 2-6.

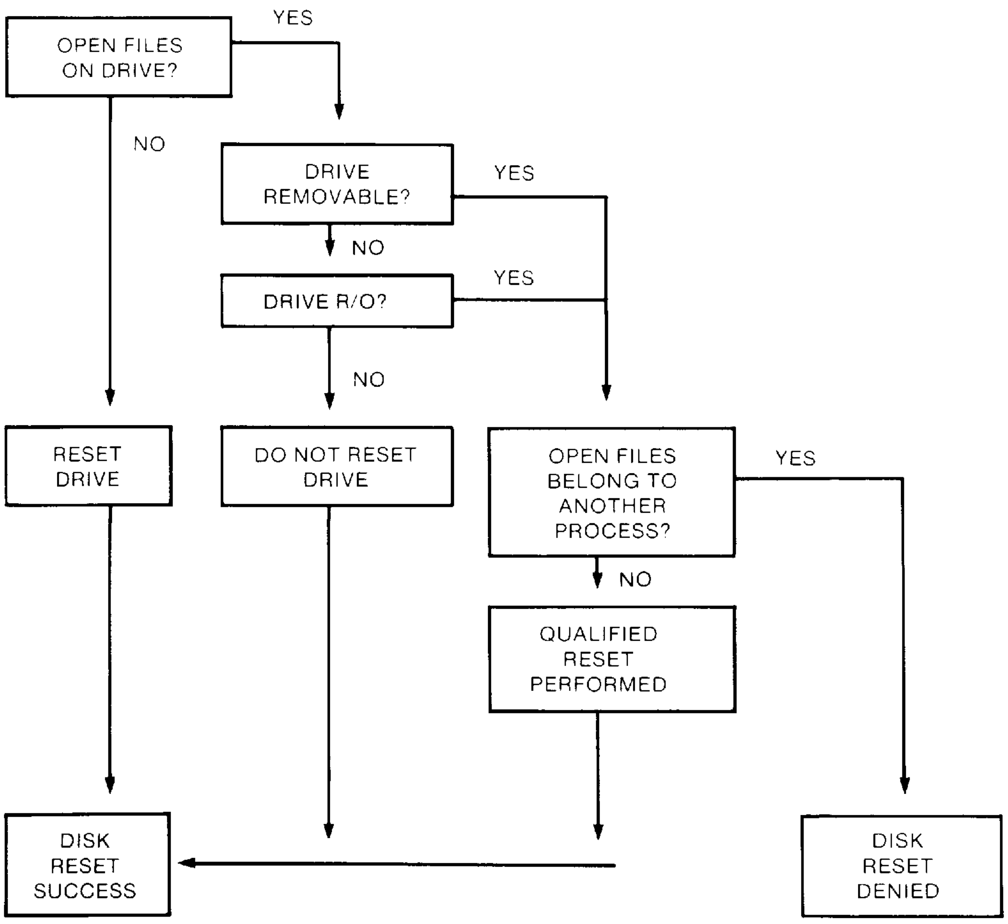

Figure 2-6. Disk System Reset

If the BDOS 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 (refer to Section 2.10 for details of FCB verification).

In all other cases where the BDOS detects a media change on a drive, the file system purges all open files on the drive from the system Lock List, and flags all processes owning a purged file for automatic open FCB verification.

Note: if a process references a purged file with a BDOS command that requires an open FCB, the file system returns to the process with an FCB checksum error.

The primary purpose of the drive reset functions is to prepare for a media change on a drive. Because a drive reset operation is conditional, it allows a process to test whether it is safe to change disks. Thus, a process should make a successful drive reset call before prompting the user to change disks. In addition, you should close all your open files on the drive, particularly files you have written to, before prompting the user to change disks. Otherwise, you might lose data.

The DRV_ACCESS and DRV_FREE system calls perform special actions under Concurrent CP/M-86. The DRV_ACCESS system call inserts a dummy open file item into the system Lock List for each specified drive. While that item exists in the system Lock List, no other process can reset the drive. The DRV_FREE system call 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 system call requiring an open FCB results in a FCB checksum error return.

The DRV_FREE system call has two important side effects. First of all, any pending blocking/deblocking buffers on a specified drive that belong to the calling process are discarded. Secondly, any data blocks that have been allocated to files that have not been closed are lost. Be sure to close your files before making this system call.

The DRV_SETRO system call is also conditional under Concurrent CP/M-86. The file system does not allow a process to set a drive to Read-Only if another process has an open file on the drive. This applies to both removable and permanent drives.

A process can prevent other processes from resetting a Read-Only drive by opening a file on the drive or by issuing a DRV_ACCESS call for the drive and then making a DRV_SETRO system call. Executing DRV_SETRO before the F_OPEN or DRV_ACCESS call leaves a window in which another process could set the drive back to Read-Write. 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.

2.18 BDOS Error Handling

The Concurrent CP/M-86 file system has an extensive error handling capability. When an error is detected, the BDOS responds in one of three ways:

1. It can return to the calling process with return codes in the AX register identifying the error.
2. It can display an error message on the console and terminate the process.
3. It can display an error message on the console and return an error code to the calling process, as in method 1.

The file system handles the majority of errors it detects by method 1. Two examples of this kind of error are the "file not found" error for the F_OPEN system call and the "reading unwritten data" error for the F_READ call. More serious errors, such as disk I/O errors, are normally handled by method 2. Errors in this category, called physical and extended errors, can also be reported by methods 1 and 3 under program control.

The BDOS Error mode, which has three states, determines how the file system handles physical and extended errors. In the default state, the BDOS displays the error message and terminates the calling process (method 2). In Return Error mode, the BDOS returns control to the calling process with the error identified in the AX register (method 1). In Return and Display Error 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).

While both return modes protect a process from termination 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 file system. Physical and extended errors are displayed on the console in the following format:

```
CP/M Error on d: error message
BDOS Function = nn File = filename.typ
```

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

The following tables detail BDOS physical and extended error messages.

Table 2-12. BDOS Physical Errors

Message	*Meaning*
Disk I/O	The "Disk I/O" 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 BDOS file-access system calls. If the XIOS read or write routine detects an error, it returns an error code to the BDOS, causing this error message.
Invalid Drive	The "Invalid Drive" 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.
Read/Only File	The BDOS returns the "Read/Only File" error message when a process attempts to write to a file with the R/O attribute set.
Read/Only Disk	The BDOS returns the "Read/Only Disk error" message when 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 DRV_SETRO system call.

Table 2-13. BDOS Extended Errors

Message	*Meaning*
File Opened in Read/Only Mode	The BDOS returns the "File Opened in Read/Only Mode" error message when a process attempts to write to a file opened in Read-Only mode. A process can open a file in Read-Only mode explicitly by setting FCB interface attribute F6'. In addition, if a process opens a file in Locked mode, the file system automatically forces the open mode to Read-Only mode when: ■ the current user number is zero and the process opens a file with the Read-Only and SYS attributes set. ■ the current user number is not zero and the process opens a user zero file with the SYS attribute set. The BDOS also returns this error if a process attempts to write to a file that is password-protected in Write mode, and it did not supply the correct password when it opened the file.
File Currently Open	The BDOS returns the "File Currently Open" error message 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 previously opened by another process or by the calling 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 F_CLOSE call.
Password Error	The BDOS returns the "Password Error" message when passwords are required and the file password is not supplied or is incorrect.

Table 2-13. (continued)

Message	*Meaning*
File Already Exists	The BDOS returns the "File Already Exists" error message for the F_MAKE and F_RENAME system calls when the BDOS detects a conflict on filename and filetype.
Illegal ? in FCB	The BDOS returns the "Illegal ? in FCB" error message when the BDOS detects a ? character in the filename or filetype of the passed FCB for the F_ATTRIB, F_OPEN, F_RENAME, F_TIMEDATE, F_WRITEXFCB, F_TRUNCATE, and F_MAKE system calls.
Open File Limit Exceeded	The BDOS returns the "Open File Limit Exceeded" error message when a process exceeds the process file lock limit specified by GENCCPM. The F_OPEN, F_MAKE, and DRV_ACCESS system calls can return this error.
No Room in System Lock List	The BDOS returns the "No Room in System Lock List" error message when no room for new entries exists within the system Lock List. The F_OPEN, F_MAKE, and DRV_ACCESS system calls can return this error.

The following paragraphs describe the error return code conventions of the file system calls. Most file system calls 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 system calls return a logical error in register AL:

F_LOCK
F_READ
F_READRAND
F_UNLOCK
F_WRITE
F_WRITERAND
F_WRITEZF

Table 2-14 lists error code definitions for register AL.

Table 2-14. BDOS Error Codes

Code	*Definition*
00H:	Function successful
01H:	Reading unwritten data
:	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
* 08H:	Record locked by another process (restricted to files opened in Unlocked mode)
09H:	Invalid FCB (previous BDOS F_CLOSE system 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 F_LOCK and F_UNLOCK system calls for files opened in Unlocked mode

For BDOS read and write system calls, the file system also sets register AH when the returned error code is a value other than zero or 0FFH. In this case, register AH contains the number of 128-byte records successfully read or written before the error was encountered. Note that register AH can only contain a nonzero value if the calling process has set the BDOS Multisector Count to a value other than one; otherwise register AH is always set to zero. On successful system calls (Error Code = 0), register AH is also set to zero. If the Error Code 0FFH, register AH contains a physical error code (see Table 2-15).

The following BDOS system calls return a directory code in register AL:

DRV_SETLABEL
F_ATTRIB
F_CLOSE
F_DELETE
F_MAKE
F_OPEN
F_RENAME
F_SIZE
F_SFIRST
F_SNEXT
F_TIMEDATE
F_TRUNCATE
F_WRITEXFCB

The directory code definitions for register AL follow.

00H - 03H	: successful function
0FFH	: unsuccessful function

With the exception of the F_SFIRST and F_SNEXT system calls, all functions in this category return with the directory code set to zero upon a successful return. However, for these two system calls, a successful directory code identifies the relative starting position of the directory entry in the calling process's current DMA buffer.

If a process uses the F_ERRMODE system call to place the BDOS in Return Error mode, the following system calls return an error flag in register AL on physical errors:

DRV_GETLABEL
DRV_ACCESS
DRV_SET
DRV_SPACE
DRV_FLUSH

The error flag definition for register AL follows.

00H	successful function
0FFH	physical error : refer to register AH

The BDOS returns nonzero values in register AH to identify a physical or extended error if the BDOS Error mode is in one of the return modes. Except for system calls that return a Directory Code, register AL equal to 0FFH indicates that register AH identifies the physical or extended error. For functions that return a Directory Code, if register AL equals 255, and register AH is not equal to zero, register AH identifies the physical or extended error. Table 2-15 shows the physical and extended error codes returned in register AH.

Table 2-15. BDOS Physical and Extended Errors

Code	*Explanation*
01H	Disk I/O Error : permanent error
02H	Read/Only Disk
03H	Read/Only File, File Opened in Read/Only Mode, or File Password Protected in Write Mode and Correct Password Not Specified
04H	Invalid Drive : drive select error
05H	File Currently Open in an incompatible mode
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

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

DRV_ALLOCVEC
DRV_DBP

When the calling process is in one of the BDOS return error modes and the BDOS detects a physical error for these system calls, 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 system calls also represent a special case.

DRV_ALLRESET
DRV_RESET
DRV_SETRO

These system calls return to the calling process with registers AL and BL set to 0FFH if another process has an open file or has made a DRV_ACCESS call that prevents the reset or write protect operation. If the calling process is not in Return Error mode, these system calls also display an error message identifying the process that prevented the requested operation.

End of Section 2

Section 3
Transient Commands

3.1 Transient Process Load and Exit

A transient process is a file of type CMD that is loaded from disk and resides in memory only during its operation. A resident system process is a file of type RSP that is included in Concurrent CP/M-86 during GENCCPM. Section 4 describes the three system memory models that determine the initial values of segment registers in transient processes.

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 system call (refer to the P_CLI system call), and passes to it the command line entered by the user. If the command is not an RSP, then the P_CLI system call locates and then loads the proper CMD file. The P_CLI system call calls the F_PARSE system call 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 P_CLI system call 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 P_CLI system call initializes the default DMA segment to the value of the initial data segment, and the default DMA offset to 0080H.

The P_CLI system call creates the new process with a P_CREATE system call and sets the initial stack so that the process can execute a Far Return instruction to terminate. A process also ends when it calls DRV_ALLRESET or P_TERM.

You can also terminate a process by typing a single CTRL-C during console input. See C_MODE for details of enabling/disabling CTRL-C. CTRL-C also forces a DRV_RESET call for each logged-in drive. This DRV_RESET operation only affects removable media drives.

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.

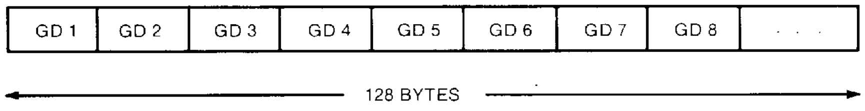

Figure 3-1. CMD File Header Format

In Figure 3-1, GD 1 through GD 8 represent group descriptors. Each group descriptor corresponds to an independently loaded program unit and has the format shown in Figure 3-2.

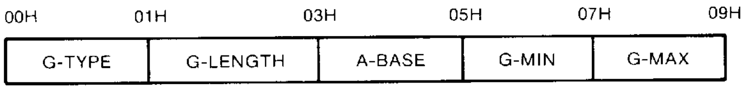

Figure 3-2. Group Descriptor Format

G_Type determines the group descriptor type. The valid group descriptors have a G_Type in the range 1 through 8, as shown in Table 3-1. All other values are reserved for future use. For a given CMD file header only a Code Group and one of any other type can be included.

If a program uses either the Small or Compact Model, the code group is typically pure; that is, it is not modified during program execution.

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

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.

Table 3-2. Group Descriptor Fields

Field	*Description*
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 (refer to 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 P_CLI and P_LOAD system calls 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 3-3.

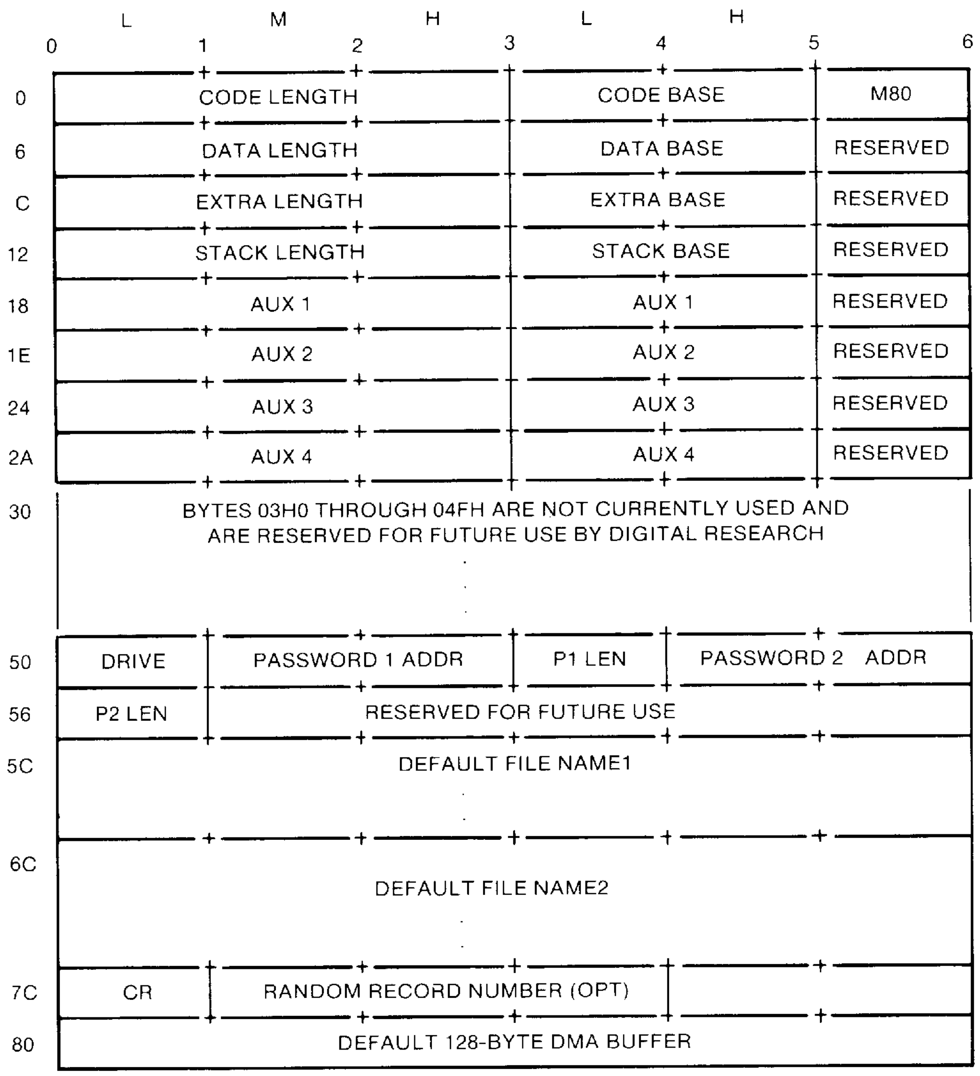
L M H L H
0 1 2 3 4 5 6
0 CODE LENGTH CODE BASE M80
6 DATA LENGTH DATA BASE RESERVED
C EXTRA LENGTH EXTRA BASE RESERVED
12 STACK LENGTH STACK BASE RESERVED
18 AUX 1 AUX 1 RESERVED
1E AUX 2 AUX 2 RESERVED
24 AUX 3 AUX 3 RESERVED
2A AUX 4 AUX 4 RESERVED
30 BYTES 03H0 THROUGH 04FH ARE NOT CURRENTLY USED AND ARE RESERVED FOR FUTURE USE BY DIGITAL RESEARCH
50 DRIVE PASSWORD 1 ADDR P1 LEN PASSWORD 2 ADDR
56 P2 LEN RESERVED FOR FUTURE USE
5C DEFAULT FILE NAME1
6C DEFAULT FILE NAME2
7C CR RANDOM RECORD NUMBER (OPT)
80 DEFAULT 128-BYTE DMA BUFFER

Figure 3-3. Concurrent CP/M-86 Base Page Values

The 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 that 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: low, middle, and high bytes.
- Base refers to the paragraph 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 P_CLI system call 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 P_CLI system call 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 P_CLI system call 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 P_CLI system call sets this field to 0 if no password is specified.
- File Name1 (bytes 005CH-0067H) is initialized by the P_CLI system call for a transient program from the first command tail operand of the command line.

- File Name2 (bytes 006CH-0077H) is initialized by the P_CLI system call for a transient program from the second command tail operand of the command line.

  Note: File Name1 can be used as part of a File Control Block (FCB) beginning at 05CH. To preserve File Name2, copy it to another location before using the FCB in file I/O system calls.
- The CR field (byte 007CH) contains the current record position used in sequential file operations with the FCB at 05CH.
- The optional Random Record Number (bytes 007DH-007FH) is an extension of the FCB at 05CH, used in random record processing.
- The Default DMA buffer (bytes 0080H-00FFH) contains the command tail when the P_CLI system call 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 most of 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, 1, 3, 4, 224, and 225, which the parent process initialized.

End of Section 3

Section 4
Command File Generation

4.1 Transient Execution Models

When the program is loaded, the initial values of the segment registers, the instruction pointer, and the stack pointer are determined by the specific type of memory model used by the transient process, indicated in the CMD file header record.

There are three memory models, the 8080 model, the Small Model, and the Compact Model, 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 that 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 can 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 P_LOAD system call 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 (P_CLI) system call 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.