CONCURRENT™ DOS 86

PROGRAMMER'S GUIDE

Version 5.0

First Edition: March 1986

This manual describes the assembly-language programming interface to the Concurrent DOS 86 operating system. It is intended as a reference manual for experienced programmers.

1066-2023-001

Contents

1 CONCURRENT DOS 86 OVERVIEW

2 THE CONCURRENT DOS 86 CP/M FILE SYSTEM

6 CONCURRENT SYSTEM CALLS

7 PC DOS SYSTEM CALLS

8 PC DOS INTERRUPT SUPPORT

9 DOS DEVICE DRIVER SUPPORT

10 WINDOW MANAGEMENT

Appendices

Figures

Tables

Listings

Foreword

Concurrent™ DOS 86 (hereinafter cited as Concurrent) is a multi- or single-user operating system targeted specifically for the Intel®8086/8088/80186/80286 family of micro-processors. It supports multiple CP/M™ or DOS[1] programming environments each implemented on a virtual console. A different task can run concurrently in each environment.

Intended audience

This manual is primarily a reference tool intended for experienced programmers. It is not a tutorial on programming. It assumes you are already familiar with general aspects of assembly-language programming and in particular, Intel microprocessor architecture. It also assumes you are familiar with the hardware components of your own system.

What's in this manual

This manual describes the invariant programming interface to Concurrent. It supports the applications programmer who wants to create software that runs in the Concurrent environment.

* Section 1 is a general overview of Concurrent.
* Section 2 describes the structure of the Concurrent's CP/M file system.
* Section 3 describes the format of transient command files.
* Section 4 describes the transient program memory models.
* Section 5 describes the creation of resident system processes.
* Section 6 describes all the generic Concurrent system calls.
* Section 7 describes all the PC DOS system calls that Concurrent supports.
* Section 8 describes Concurrent's support for PC DOS interrupts.
* Section 9 describes PC DOS driver support.
* Section 10 describes Window management.

---

[1] In this manual, DOS refers to both PC DOS and MS-DOS

Where to find more information

The Concurrent DOS User's Guide, (hereinafter cited as the User's Guide) documents Concurrent's user interface, explaining the various features used to execute application programs and Digital Research utility programs.

The Concurrent DOS Reference Guide, (cited as the Reference Guide) is a detailed reference manual that describes all of Concurrent's commands.

The Concurrent DOS 86 System Guide, (cited as the System Guide) documents Concurrent's internal, hardware-dependent structures.

Two other documents describe Digital Research software that you can use to write, debug, and verify software written for the Concurrent environment.

RASM-86™, relocating assembler, LINK-86™, linkage editor, and LIB-86™, software librarian are described in Programmer's Utilities Guide for the CP/M-86... Family of Operating Systems, (cited as the Programmer's Utilities Guide).

SID-86™, symbolic instruction debugger is described in SID-86 Productivity Tool User's Guide, (cited as SID-86 User's Guide).[2]

Notation conventions

The following notation conventions are used throughout this manual:

n	A numeric value indicates a decimal number unless otherwise stated.
nH	A numeric value followed by the capital letter H indicates the number is a hexadecimal value.
. . .	Horizontal ellipses indicate the immediately preceding item can occur once, or any number of times in succession. Vertical ellipses indicate an omitted portion of a source program or example; only the relevant part is shown.
CTRL	In the text, the symbol CTRL represents a control character. Thus, CTRL-C means control-C. In any listing that shows example console interaction, the symbol ^ is the echo of a control character.

[2]RASM-86, LINK-86, LIB-86 and SID-86 are sold together as DR Assembler Plus Tools™

CONCURRENT DOS 86 OVERVIEW

1.1 Introduction

Concurrent DOS 86 is a multi- or single-user, multitasking operating system. It lets you run multiple programs simultaneously by initiating tasks on two or more terminals or virtual consoles. Application programs have access to system calls used by Concurrent to control the multiprogramming environment. Concurrent supports extended features, such as communication among and synchronization of independently running processes. Figure 1-1 depicts the relationships between application programs, virtual environments, virtual consoles, and the user's physical terminal.

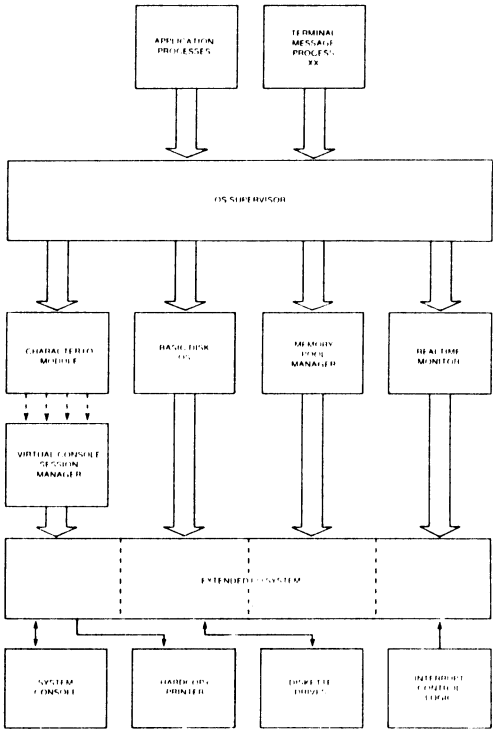


Figure 1-1. Concurrent DOS 86 Virtual/Physical Environments

In the Concurrent 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 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 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 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 made an integral part of Concurrent during system generation, so they are immediately available to perform operating system tasks. For example, the CLOCK process is an RSP that maintains the time of day within Concurrent.

The following list briefly summarizes Concurrent's capabilities:

* Interprocess communication, synchronization, and mutual exclusion functions are provided by system queues.
* A logical interrupt mechanism using flags allows Concurrent to interface with any physical interrupt structure.
* System timing functions enable processes to compute elapsed times, delay execution for specified intervals, and to access and set the current date and time.
* The shared file system allows multiple programs to access common data files while maintaining data integrity.
* Ability to run DOS programs by providing software emulation of DOS system calls.
* Shared code support eliminates loading multiple copies of the same program and conserves memory space.
* 8087 support takes advantage of fast 8087 math instructions.
* Support for memory paging hardware allows memory to be expanded up to 8 megabytes.
* 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.

Concurrent is composed of the following modules:

* The **Supervisor** module (SUP) 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 **Real-time Monitor** module (RTM) monitors the execution of running processes and arbitrates conflicts for the system's resources.
* The **Memory Management** module (MEM) allocates and frees memory upon demand from executing processes.
* The **Basic Disk Operating System** (BDOS) is the hardware-independent module that contains the logically invariant portion of the file system. The BDOS file system is explained in detail in Section 2.
* The **Character I/O** module (CIO) handles all character I/O for console, list, and auxiliary devices.
* The **Virtual Console Screen Manager** extends the CIO to support virtual console environments.
* The **Extended I/O System** (XIOS) is the hardware-dependent module that defines Concurrent's interface to a specific hardware environment. See the System Guide for more detailed information about the XIOS.

1.2 Supervisor (SUP)

The Supervisor module (SUP) manages the interface between processes and Concurrent's multitasking nucleus. It also manages internal communication between the other Concurrent 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) calls.

1.3 Real-time Monitor (RTM)

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

1.3.1 Process Dispatching

Although Concurrent 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. 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.

Every 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 priority order. Suspended processes are on other system lists, depending on why the processes were suspended.

Process dispatching 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, the Dispatcher places its Process Descriptor on the **Delay List**. When a process releases a resource, the Dispatcher usually places the process back on the Ready List. If another process is waiting for the resource, the Dispatcher removes that process from its current system list and places it on the Ready List.
3. The Dispatcher chooses the highest priority process on the Ready List 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 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, list, or auxiliary 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 is a **priority-driven** system, which means that during a dispatch, the Dispatcher gives the CPU resource to the process with the best priority. The Dispatcher allots equal shares of Concurrent's resources to processes with the same priority. With priority dispatching, the Dispatcher never gives 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.

When Concurrent 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.

1.3.2 Queue Management

Queues perform several critical functions for processes running under Concurrent. A process can use a queue for communicating with another process, synchronizing its execution with that of another process, and for excluding 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 implements these special data structures as memory files that contain room for a specified number of fixed-length messages.

When the Q_MAKE call creates a queue, the 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, Concurrent returns an Error Code to the calling process. However, if a process attempts an unconditional queue operation in these circumstances, Concurrent 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, Concurrent 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. 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 Concurrent 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'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. The CLOCK process issues DEV_WAITFLAG 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 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 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 call, it specifies how many ticks it should be suspended from execution. Concurrent 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 Dispatcher 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 P_DELAY in Section 6 describes how to determine the correct number of ticks to delay 1 second.

1.4 Memory Management Module (MEM)

Concurrent supports an extended, fixed partition model of memory management; the Memory Module handles all memory management system calls. In practice, the exact method that Concurrent 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-specific memory system calls described in Section 6.

1.4.1 Expanded Memory Support

Concurrent also supports Expanded Memory Management (EMM), so a process can address other memory not currently available in the processor's normal 1Mb address space. Concurrent's EMM support is generic and hardware-independent.

Concurrent supports EMM through the Memory Manager (MEM), Real-time Monitor (RTM), and the XIOS functions that enable it to perform a technique called **memory paging**. Concurrent dynamically maps regions (usually 16K bytes) of physical memory in and out of the 1Mb logical address space accessible to the processor. The regions of physical memory are called **pages**, and the areas in the processor's logical address space into which pages are mapped are called **logical address windows**, or simply windows.

1.4.2 Memory Paging Environments

Concurrent supports EMM in two dissimilar environments. The first environment is generic. In this environment, the XIOS is responsible for managing the memory mapping hardware and allocating physical pages of memory.

The second environment is that of an IBM® Personal Computer (or compatible) with an add-on memory board such as the AST™ RAMpage!™. Concurrent also supports any board that conforms to the Intel®/Lotus™ Above Board standard, although such boards do not provide memory paging. In this environment, certain applications call the EMM Driver to perform their own memory management tasks completely independent of Concurrent. This second environment is called the "EMM environment."

Generic Environment

In the generic environment, the Supervisor and applications call the Memory Manager (MEM) for all memory requests. MEM passes these calls to the XIOS as memory page allocation and release requests. During process dispatching, the Real-time Monitor (RTM) generates XIOS calls to save the current state of the memory mapping hardware and to restore a previously saved state.

Figure 1-2 illustrates the generic environment's interfaces.

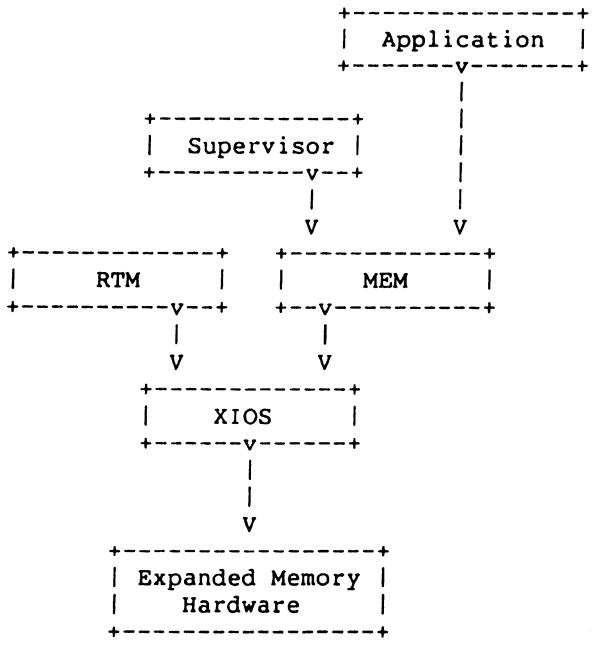


Figure 1-2. Generic Environment Memory Paging Interfaces

EMM Environment

In the EMM environment, the XIOS passes all memory page allocation and release requests to the EMM Driver, which handles the page mapping hardware. Calls into the XIOS are translated to EMM Driver calls, and the EMM Driver is invoked with an Interrupt 67H.

An application running in the EMM environment can make calls directly to the EMM Driver, but the **Interceptor** module intercepts such calls so Concurrent can handle context switching and memory deallocation for aborted processes. To reserve some memory for system use, the Interceptor might not allow an application to know the total number of available memory pages.

The Interceptor's primary functions are creating and linking new **Memory Page Allocation Descriptors** (MPADs) as an application performs EMM Allocate calls. The Interceptor destroys these MPADs when the application performs an EMM Close call. These functions allow the Interceptor to track an application's calls to the EMM Driver. The MPAD data structure is described in the System Guide.

Figure 1-3 illustrates the interfaces within the EMM environment.

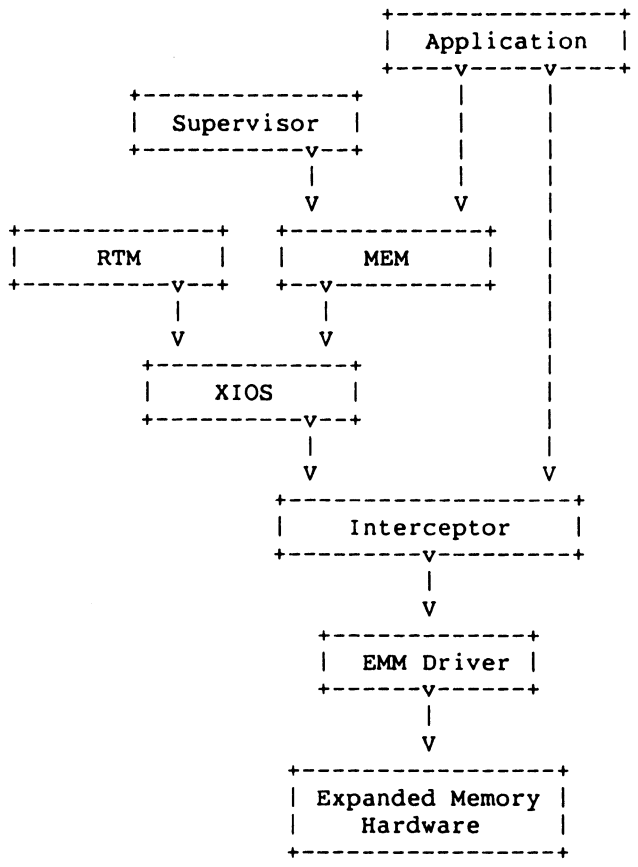


Figure 1-3. EMM Environment Memory Paging

1.4.3 EMM Drivers

Memory paging hardware requires a corresponding Expanded Memory Management (EMM) software driver. Section 9 contains information about configuring EMM drivers. See the System Guide for more information about Concurrent's XIOS support for Expanded Memory Management.

1.5 Basic Disk Operating System (BDOS)

Concurrent's BDOS is an upward-compatible version of the single-tasking CP/M-86 BDOS. It handles file creation and deletion, performs sequential or random file access, and allocates and frees disk space.

Concurrent's BDOS is modified to accept both CP/M and DOS media files, and extended to provide support for multiple virtual consoles and list devices as well as those services required in a multitasking environment. The major extensions to the file system are:

* File locking. Files opened under Concurrent cannot be opened or deleted by other processes.
* Shared access to files. As a special option, independent users can open the same file in Shared or Unlocked mode. Concurrent supports record locking and unlocking commands for files opened in this mode and protects files opened in Shared mode from deletion by other processes.
* 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.[1]
* Password Protection. The BDOS password protection feature is optional at either the file or drive level. The user or applications program assigns disk drive passwords, while application programs can assign file protection passwords in several modes.[1]
* Extended Error Module. Besides the default error mode, Concurrent 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, list, and auxiliary device I/O. 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.

---

[1] CP/M media only

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 System Guide for more information about LCBs and CCBs.

1.7 Virtual Console Screen Management

Virtual console screen management is coordinated by three separate modules: the CIO, the PIN (Physical INput) process, 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 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 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 Concurrent where 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 Process. The DRV_RESET calls affect only the removable media drives, as specified in the CKS field of the Disk Parameter Blocks in the XIOS (see the System Guide for 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 the 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 additional primitive system calls, such as DEV_POLL, support Concurrent's additional features, including elimination of wait loops for real-time I/O operations.

1.9 Terminal Message Processes (TMP)

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, a transient program is one that is not system-resident. Concurrent must load such a program from disk into available memory every time it executes. The command file of a transient program is identified by the filetype CMD. When you enter a command at the console, Concurrent searches on disk for the appropriate CMD file, loads it, and initiates it.

Concurrent 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 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 preserves the contents of registers SI, DI, BP, SP, SS, DS, and CS through the 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 can show Concurrent's internal state including memory allocation, current processes, system queue activity, and many informative parameters associated with these system data structures.

SYSTAT can present two views: either a static snapshot of system activity, or a continuous, real-time window into Concurrent.

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)
  C(onsoles)
  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:

```
SYSTAT [option]
SYSTAT [option C]
SYSTAT [option C ##]

-where-

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

->  option =

    M(emory) P(rocesses) O(verview) C(onsols)
    U(ser Processes) Q(ueues) H(elp)


Type any key to leave and return to the main menu.
```

The M, P, Q, and U and C options ask you if you prefer a continuous display. If you type y, Concurrent 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 **O(verview)** option displays an overview of the system parameters, as specified at system generation time. The display is not continuous.

The **P(rocesses)** 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 **C(onsoles)** option displays console information; for example, background, foreground, buffered, suspended, purging, and CTRL-Q.

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

End of Section 1

THE CONCURRENT DOS 86 CP/M FILE SYSTEM

2.1 File System Overview

The Basic Disk Operating System (BDOS) file system supports from one to thirteen 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, which 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, Concurrent's 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 process making 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** whereby question marks and asterisks are used 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 calls and drive-related calls. The file-access calls have mnemonics beginning with F_, and the drive-related 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 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, F_CLOSE 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. A drive reset call prepares for a media change on drives that support removable media. Under Concurrent, drive reset calls are conditional. A process cannot reset a drive if another process has a file open on the drive.

Table 2-1 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
F_RANDREC	Return Record Number For File Read-Write
F_READ	Read Record Sequentially From File

Table 2-1. (Cont'd)

Mnemonic	Description
F_READRAND	Read Random Record From File
F_RENAME	Rename File
F_SETDATE	Set File Time and Date Stamp
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, a file specification has 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 is a letter (A,B,C, etc.), where the actual drive letters supported on a given system are determined by the XIOS. When no drive letter is specified, Concurrent assumes the current default drive.

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.

The drive, type, and password fields are optional, and the delimiters : . ; are required only when specifying their associated fields. Omitting the optional type or password fields implies a field specification of all blanks.

Under Concurrent, the P_CLI call interprets ASCII command lines and loads programs. P_CLI in turn calls F_PARSE 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

F_PARSE also excludes all control characters from the file specification fields and translates all lowercase letters to uppercase.

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???????.???. F_SFIRST, F_SNEXT, and F_DELETE all 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 calls treat a question mark in the filename or filetype fields as an error.

It is not mandatory to follow Concurrent's file naming conventions 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, P_CLI cannot locate a command file in the directory if its filename or filetype field contains a lowercase 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
BAK	Text or Source Back-up
BAS	BASIC Source File
BAT	DOS Submit File
C	C Source File
CMD	8086 Command File (CP/M)
COM	8086 Command File (DOS)
CON	Concurrent Module
DAT	Data File
EXE	8086 Command File (DOS)
INT	Intermediate File
LIB	Library File
L86	Library File
LST	List File
PLI	PL/I Source File
RSP	Resident System Process
SYM	Symbol File
SYS	System File
$$$	Temporary File

2.3 Disk Drive and File Organization

The file system can support up to thirteen logical drives, identified by the letters A through M. 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 may consist of a set of up to 262,144 (40000H) 128-byte records. Each record is identified by its position in the file, which 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 (on CP/M media files only.)

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 be created and accessed only 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 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 utilities such as TYPE and RASM-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 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. Some file-access system calls use a modified FCB to perform special DOS-directory related operations when accessing DOS media files.

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 F_READRAND, F_WRITERAND, F_WRITEZF, F_LOCK, F_UNLOCK, F_RANDREC, F_SIZE, and F_TRUNCATE, the minimum FCB length is 36 bytes. When F_OPEN or F_MAKE open a file in Unlocked mode, the FCB must be at least 35 bytes long.

Figure 2-1 shows the FCB data structure.

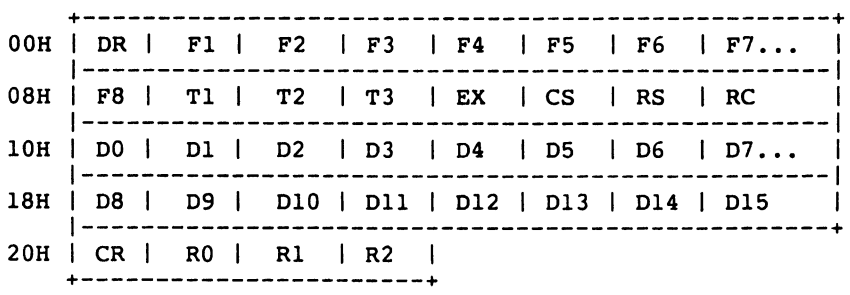


Figure 2-1. FCB - File Control Block

The fields in the FCB are defined as follows:

Table 2-5. FCB Field Definitions

Field	Definitions
DR	Contains the Drive Code, with 0 for the default drive, 1 for drive A, 2 for drive B, etc. Note that drives N, O, and P are reserved for Concurrent.
F1...F8	Contain the filename in ASCII uppercase. The high-order bits of F1...F8 are called attribute bits (see Table 2-8 and Table 2-12).
T1...T3	Contain the filetype in ASCII uppercase. The high-order bits of T1...T3' are called attribute bits (see Table 2-7).
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.
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 and reserved for system use. Also used to specify the new filename and filetype with F_RENAME.
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 (see Section 2.10).

2.4.1 FCB Initialization and Use

The calling process must initialize bytes 0 through 11 of the referenced FCB before calling F_ATTRIB, F_DELETE, F_MAKE, F_OPEN, F_RENAME, F_SFIRST, F_SIZE, F_SNEXT, F_TIMEDATE, F_TRUNCATE, or F_WRITEXFCB. Normally, the DR field specifies the file's drive location, and the name and type fields specify file's name.

You must also set the EX field of the FCB before calling F_MAKE, F_OPEN, F_SFIRST, or F_WRITEXFCB. Except for F_WRITEXFCB, you can usually set this field to zero. Note that F_RENAME 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, F_SNEXT expects an FCB initialized by a prior F_SFIRST call. In addition, F_LOCK, F_READ, F_READRAND, F_UNLOCK, F_WRITERAND, and F_WRITEZF all require an FCB that has been activated for record operations. Under Concurrent, only F_OPEN and F_MAKE can activate an FCB.

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

F_SFIRST, F_SNEXT, and F_DELETE 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 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 F_OPEN and F_MAKE 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 illegally modified by the calling process. Thus, all read, write, lock, and unlock operations on a file must specify a valid activated FCB; otherwise, the BDOS returns a checksum error 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 2.4.4). 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, F_CLOSE 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 FCB Initialization for DOS Media Files

When a process calls F_CLOSE, F_DELETE, F_MAKE, F_OPEN, F_SFIRST, or F_SNEXT with the high-bit of the FCB's drive specifier byte turned on (logically ORed with 080H), the call performs the appropriate DOS directory-related operation as follows:

F_CLOSE	Forces the FCB-specified drive to the root directory.
F_DELETE	Deletes a subdirectory in relation to the current directory.
F_MAKE	Creates a subdirectory in relation to the current directory.
F_OPEN	Opens a subdirectory in relation to the current directory.
F_SFIRST	Finds the first matching DOS directory FCB when the high-bit of the referenced FCB's drive specifier byte is set.
F_SNEXT	Finds the next DOS directory FCB that matches the FCB (byte 0 high-bit set) specified in a previous F_SFIRST call.

Figure 2-2 shows an FCB initialized for directory operations with DOS media files.

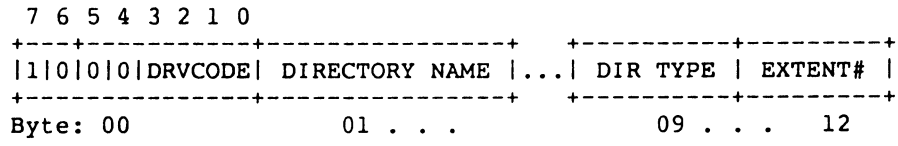


Figure 2-2. FCB Initialized for a DOS Directory

The calling process sets the high-order bit of byte 0 in the FCB and places the drive code in the remainder of byte 0. Bytes 01 through 08 are initialized with the directory name, and bytes 09 through 11 contain the directory type field.

The calling process initializes the Extent field (byte 12) to a value of 0, or sets it to a floating-drive code. To specify a floating drive, set byte 12 to 1 for N, 2 for O, or 3 for P. This will map the appropriate floating drive to the drive and directory specified in the FCB.

Note that when a process calls F_SFIRST (11H) or F_SNEXT (12H) to locate a DOS directory FCB, it must set the first bit of the referenced FCB. Concurrent clears this bit following these calls. Without the first bit of the referenced FCB set on F_SFIRST and F_SNEXT calls, Concurrent will not search for any DOS directory FCBs.

The first two bytes of the Disk Map field in the FCB (beginning at offset 10H) have special meaning when a process reads or writes to DOS media. Table 2-6 shows the values these bytes have for DOS media file I/O.

Table 2-6. FCB Disk Map Values for DOS Media

Byte	FCB Offset	Bit Values
D0	10H	Bit 0-4 Reserved Bit 5 1 for Hidden File Bit 6 1 for Subdirectory Bit 7 Always Set
D1	11H	Same as D0

Bytes D6 through D9 (16H-19H) also have special meaning for processes reading from or writing to files on DOS media. These bytes contain the time of day and date the DOS media file was created or last updated. This information is mapped to the bits of D6-D9 as shown in Figure 2-3.

```
 h  h  h  h  h  m  m  m  m  m  m  s  s  s  s  s
15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

<--------- D6 --------> <-------- D7 --------->

      hh = 00-23, mm = 00-59, ss = 00 - 59
      hours       minutes     seconds

 Y  Y  Y  Y  Y  Y  Y  M  M  M  M  D  D  D  D  D
15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

<--------- D8---------> <-------- D9 --------->

 DD = 01-31, MM = 01-12, YY = 0-119 (1980-2099)
 day         month       year
```

Figure 2-3. FCB Time and Date Fields for DOS Files

You can perform the following steps to determine the type of medium in the current default drive:

1. Call F_SFIRST with byte 0 of the referenced FCB set to a question mark. This returns the first FCB in the directory.
2. Read byte 0 of the first directory FCB to identify the directory label. If this byte has a value of 32 (20H), the returned FCB is a directory label. Any other value indicates CP/M media.
3. Read byte 0FH of the directory label; a value of 80H indicates DOS media. Any other value indicates CP/M media.

2.4.3 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.

Attributes F1' through F4' of command files are defined as **Compatibility** attributes (see Section 2.12). For all other files, F1' through F4' are available for user definition. Attributes F5' and F6' are defined as **Interface** attributes (see Section 2.4.4).

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

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

Table 2-7 shows the definitions for file attributes T1',T2', and T3'.

Table 2-7. File Attribute Definitions

Attribute	Definition
T1': Read-Only Attribute	Attribute T1', if set, prevents write operations to a file.
T2': System Attribute	Attribute T2', if set, identifies the file as a Concurrent System file. Concurrent's 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	Attribute T3' supports user-written archive programs. 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 F_SFIRST and F_SNEXT. If all directory entries have the archive attribute set, the file has not been modified since the previous archive. The Concurrent PIP utility supports file archival.

2.4.4 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 2-8 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 2-8. 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

Concurrent 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, and 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.

However, it is possible for a process to access files on different user numbers by setting the user number to the file's user number with an F_USERNUM 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 P_CLI initiates a transient process or Resident System Process (see Section 5), it sets the user number to the default value established by the process issuing the P_CLI call. The sending process is usually the Terminal Message Process (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 an 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. 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 manager control over which files are directly 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.

Figure 2-4 shows the format of the directory label.

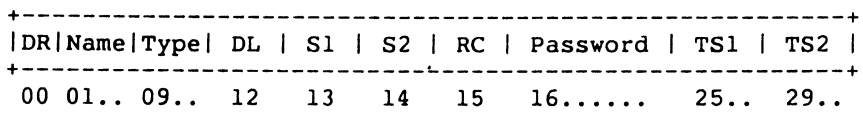


Figure 2-4. Directory Label Format

Table 2-9 defines the fields in the directory label.

Table 2-9. 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 DRV_SETLABEL to create a directory label or update its fields. DRV_SETLABEL 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: Concurrent does not provide a system call to read the directory label FCB directly. However, you can read the directory label data byte directly with the DRV_GETLABEL call. In addition, you can use the search 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. Figure 2-5 shows the format of the XFCB.

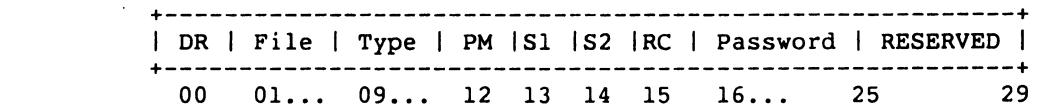


Figure 2-5. XFCB - Extended File Control Block

Table 2-10 defines the fields in the XFCB.

Table 2-10. 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

You can create an XFCB only 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 F_MAKE or F_WRITEXFCB. F_MAKE creates an XFCB if the calling process requests that a password be assigned to the created file. F_WRITEXFCB 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: with F_MAKE or with F_WRITEXFCB. You can also change a file's password or password mode with F_WRITEXFCB 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 BDOS provides password protection in one of three modes when password support is enabled by the directory label. Table 2-11 shows the difference in access level allowed to BDOS system calls when the password is not supplied.

Table 2-11. Password Protection Modes

Mode	Access Allowed Without Password
Read	File cannot be read, modified, or deleted.
Write	File can be read, but not modified or deleted.
Delete	File 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, mode 1 protection implies mode 2 and 3 protection, and mode 2 protection implies mode 3 protection. All three modes require you 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, F_SFIRST and F_SNEXT 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 an F_PASSWD 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 FSET utility; all programs loaded by this TMP inherit the same default password.

2.8 File Date and Time Stamps: SFCBs

The Concurrent 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 2-6

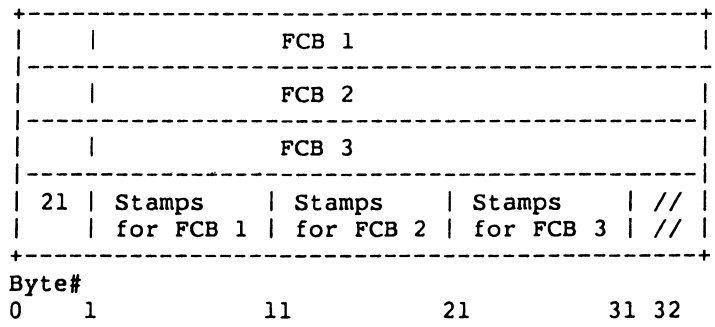


Figure 2-6. Directory Record with SFCB

Figure 2-6 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 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 2-7 shows the detail of the SFCB subfields.

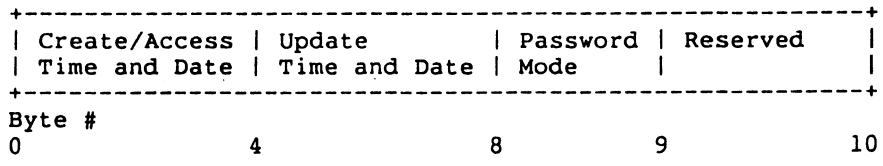


Figure 2-7. SFCB Subfields

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 calls. In addition, you can make a F_TIMEDATE call to retrieve the date and time stamps and password mode of a specified file. The explanation of T_GET in Section 6 describes the format of a date and time stamp field.

Concurrent 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 (see the User's Guide for details of using DATE).

2.9 File Open Modes

The file system provides three different modes for opening files:

Locked Mode

Locked mode is the Default mode for opening files under Concurrent. 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.

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. Unlocked 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 F_OPEN 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 calls which 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.

F_OPEN 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, F_OPEN 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, with the open mode automatically forced to Read-Only.

F_OPEN 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 both 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 to prevent sharing the file with other processes.

F_OPEN and F_MAKE both use FCB interface attributes F5' and F6' to specify the open mode. The interface attribute definitions for these functions are listed in Table 2-8.

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

2.10 File Security

The security measures implemented in the file system are designed to prevent accidental collisions between running processes. It is not possible to provide total security because the file system maintains file allocation information in open FCBs in the user's memory region, and Concurrent 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, and 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 calls compute and assign checksums to FCBs. F_READRAND, F_READ, F_WRITERAND, F_WRITEZF, F_WRITE, F_LOCK, and F_UNLOCK subsequently verify and recompute the checksums when they change the FCB. F_CLOSE also verifies FCB checksums. Although you can disable FCB verification by these system calls (see Section 2.12), it is not recommended because Concurrent's file security is reduced.

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 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 Lock List 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 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 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 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 Lock List is limited. If a process attempts to open a file and no space remains in the 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 an F_ERRMODE call. Note that the size of the Lock List and the process open file limit are GENCCPM parameters.

Under Concurrent, deleting an open file is not recommended 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 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 BDOS performs FCB verification whenever it purges open file entries from the system Lock List in the following situations:

* A process makes a F_ATTRIB, F_DELETE, F_RENAME, or F_TRUNCATE 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.

2.11 Extended File Locking

Extended file locking enables a Concurrent 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. F_CLOSE 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, F_CLOSE 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 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 F_DELETE, 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 an 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 steps illustrate 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 F_RENAME 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 an F_ATTRIB 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 are defined as file attributes F1' through F4' of program (CMD) files, and they provide a mechanism to modify some of Concurrent's file security rules. This facility is needed because some programs developed under earlier Digital Research operating systems do not run properly under Concurrent. Most of the problems encountered by such programs occur because they were designed for single-tasking operating systems where file security is not required.

For example, suppose a program closes a file and then continues reading and writing to the file. Under CP/M-86, this is not a problem but under Concurrent, the BDOS intercepts open file operations with a deactivated FCB to ensure file system integrity. Compatibility attributes are a tool for dealing with such situations, and you should use them only with existing programs that run properly under CP/M-86, not with new programs developed under Concurrent.

If the GENCCPM COMPATMODE option has been selected during system generation, you can use Concurrent's FSET utility to set compatibility attributes from the command line. When COMPATMODE is selected, the P_CLI call interrogates the command file's compatibility attributes during program loading and modifies the Concurrent file security rules for the loaded program.

Table 2-12 defines the Concurrent BDOS Compatibility Attributes.

Table 2-12. 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, 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. The F1' 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, the F1' 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 the F1' 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.
F2'	Change F_CLOSE to partial close. Processes running with F2' set, only make partial F_CLOSE calls. The F2' 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 to set the Lock List parameters to higher values when you generate the system with GENCCPM.

Table 2-12. (Cont'd)

Attribute	Definition
F3'	Ignore close checksum errors. The F3' attribute changes the way F_CLOSE handles Close Checksum errors. Normally, the file system prints an error message at the console and terminates the calling process. However, if F3' is set, F_CLOSE 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 F4' also sets attributes F2' and F3'. The F4' 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's file security.

Use Concurrent's FSET utility to specify the combination of compatibility attributes you want set in the program's command file. For example,

```
A>FSET filespec [f1=on]

A>FSET filespec [f1=on,f3=on]

A>FSET filespec [f4=on]
```

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

* 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 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 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, using 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 with F_MULTISEC. P_CLI 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 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, F_WRITE, F_READRAND, F_WRITERAND, F_WRITEZF, F_LOCK and F_UNLOCK. If the Multisector Count is N, making one of these calls is equivalent to making N 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 F_LOCK and F_UNLOCK.

2.14 Concurrent File Access

Concurrent 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'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.

F_LOCK and F_UNLOCK allow a process to establish and release temporary ownership of 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 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.

F_LOCK 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.

F_LOCK 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, if a process attempts to convert a record's shared lock to an exclusive lock if another process has a shared lock on the record, Concurrent returns an error.

F_LOCK has another option, specified by interface attribute F6', which controls whether a record must exist in order to be locked. If you make an F_LOCK call with F6' = 0, the file system returns an error code if the specified record does not exist. Setting F6' to 1 requests a logical lock operation. Logical lock operations are only limited by the maximum Concurrent 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.

F_UNLOCK is similar to F_LOCK 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.

F_UNLOCK has one option, controlled by interface attribute F5'. If F5' is set to one, F_UNLOCK 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 F_CLOSE also removes all locks for a file on permanent close operations.

If the BDOS Multisector Count is greater than one, F_LOCK and F_UNLOCK 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, F_LOCK also returns an error code if another process has an exclusive lock on any record within the sequence.

When a process makes an F_LOCK 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 call removes the locked entry from the list. While the lock entry exists in the 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. The file system performs no action and returns an error code if insufficient available entries exist in the 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 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 F_SIZE call returns the last Random Record Number, + 1, of the last record of a file.

The F_ATTRIB call can set a file's byte count. This is an option controlled by interface attribute F6'. Conversely, F_OPEN can return a file's byte count to the CR field of the FCB. F_SFIRST and F_SNEXT also return a file's 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 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, 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 physical record size is not restricted to 128 bytes, but can range from 128 bytes to 4K bytes. 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** and is required in write operations. The reverse process of breaking up physical records into their component 128-byte logical records is called **record deblocking** and is required in read operations. Under Concurrent, 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, 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, you can use F_FLUSH 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 an F_FLUSH call in the F_CLOSE call. Thus, it is sufficient to make an F_CLOSE 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 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, 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 DRV_ALLRESET or DRV_RESET 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 DRV_ALLRESET and DRV_RESET have similar effects except that DRV_ALLRESET affects all drives on the system. You can specify the combination of drives to reset with DRV_RESET.

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, 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 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 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.

Figure 2-8 illustrates the logic flow of the drive reset operation.

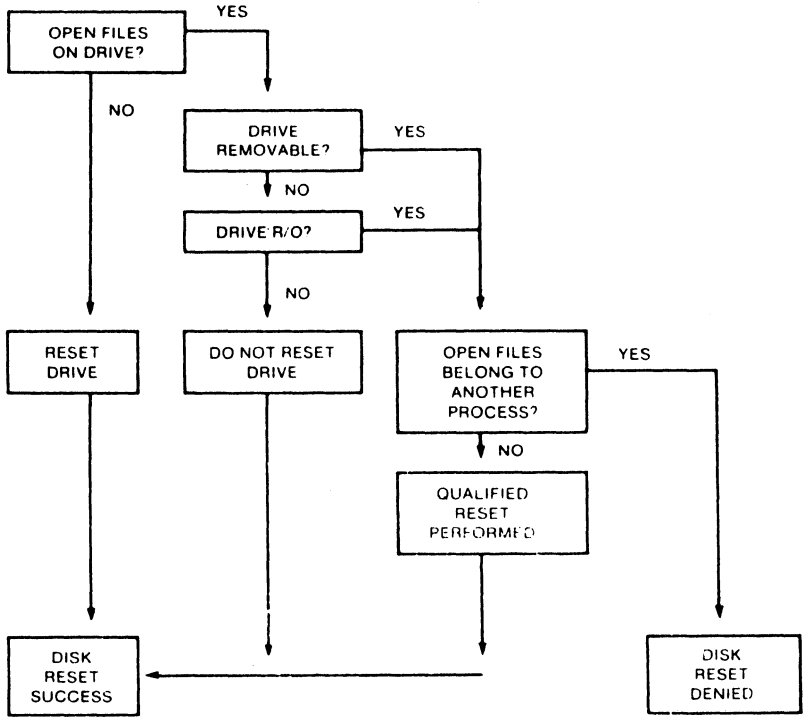


Figure 2-8. Disk System Reset

If the BDOS detects a media change on a drive after a qualified reset, it purges all open files on the drive from the 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 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.

DRV_ACCESS and DRV_FREE perform special actions under Concurrent. DRV_ACCESS 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. DRV_FREE 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.

DRV_FREE 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 calling DRV_FREE.

DRV_SETRO is also conditional under Concurrent. 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 call. Executing DRV_SETRO before the F_OPEN or DRV_ACCESS call leaves an interval 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 Lock List, no other process can reset the drive to take it out of Read-Only status.

2.18 BDOS Error Handling

The Concurrent file system has an extensive error handling capability. When an error is detected, the BDOS can respond in one of three ways:

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

The file system handles the majority of errors it detects by method 2. Two examples of this kind of error are the "file not found" error for F_OPEN and the "reading unwritten data" error for F_READ.

More serious errors, such as disk I/O errors, are normally handled by method 1. Errors in this category, called physical and extended errors, can also be reported by methods 2 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** Error mode, the BDOS displays the error message and terminates the calling process (method 1).
* In **Return** Error mode, the BDOS returns control to the calling process with the error identified in register AX (method 2).
* In **Return and Display** Error mode, the BDOS returns control to the calling process with the error identified in register AX 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 in the following format:

```
Concurrent 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 2-13. 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 DRV_SETRO.

Table 2-14. 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 System attributes set. * the current user number is not zero and the process opens a user zero file with the System 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 an 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 2-14. (Cont'd)

Message	Meaning
File Already Exists	The BDOS returns the "File Already Exists" error message for the F_MAKE and F_RENAME 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 F_ATTRIB, F_OPEN, F_RENAME, F_TIMEDATE, F_WRITEXFCB, F_TRUNCATE, and F_MAKE.
Open File Limit Exceeded	The BDOS returns the "Open File Limit Exceeded" error message when a process exceeds Concurrent's process file lock limit. F_OPEN, F_MAKE, and DRV_ACCESS 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 Lock List. F_OPEN, F_MAKE, and DRV_ACCESS 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 an error code in register AL:

* F_LOCK
* F_READ
* F_READRAND
* F_UNLOCK
* F_WRITE
* F_WRITERAND
* F_WRITEZF

Table 2-15 lists error code definitions for register AL.

Table 2-15. BDOS Logical Errors

Code	Definition
00H	Function successful
01H	Reading unwritten data or No available directory space on (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 (only for files opened in Unlocked Mode)
09H	Invalid FCB (error in previous F_CLOSE call
0AH	FCB checksum error
0BH	Unlocked file unallocated block verify error (only for files opened in Unlocked Mode)
0CH	Process record lock limit exceeded (returned only by F_LOCK and F_UNLOCK for files opened in Unlocked mode
0DH	Invalid File ID (returned only by F_LOCK and F_UNLOCK for files opened in Unlocked mode
0EH	No room in System Lock List (returned only by F_LOCK and F_UNLOCK for files opened in Unlocked mode
0FFH	Physical error : refer to register AH

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 2-16).

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_SETDATE
* F_SIZE
* F_SFIRST
* F_SNEXT
* F_TIMEDATE
* F_TRUNCATE
* F_WRITEXFCB

The directory code definitions for register AL are:

00H - 03H	successful function
0FFH	unsuccessful function

With the exception of F_SFIRST and F_SNEXT, 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 F_ERRMODE 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 definitions for register AL are:

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 0FFH, and register AH is not equal to zero, register AH identifies the physical or extended error. Table 2-16 shows the physical and extended error codes returned in register AH.

Table 2-16. 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, 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

TRANSIENT COMMAND FILES

3.1 Transient Program Loading

A **transient** program is a file of type CMD that is loaded from disk and resides in memory only during its operation. A **Resident System Process** (RSP) is a file that is included in Concurrent during system generation.

You can initiate a transient process by entering a command at a system console. The console's TMP (Terminal Message Process) then calls P_CLI (Command Line Interpreter), and passes to it the command line you entered. If the command is not an RSP, then P_CLI locates and then loads the proper CMD file. P_CLI then calls F_PARSE to parse up to two filenames following the command, and place the properly formatted FCBs at locations 005CH and 006CH in the Base Page of the initial Data Segment (see Section 3.3).

P_CLI initializes memory, the Process Descriptor (PD), and the User Data Area (UDA), and then allocates a 96-byte stack area, independent of the program, to contain the process's initial stack. If 8087 processing is required (see Section 3.1.2) P_CLI allocates an additional 96 bytes for the UDA.

Concurrent divides the **Direct Memory Address** (DMA) into the DMA segment address and the DMA offset. P_CLI initializes the default DMA segment to the value of the initial data segment, and the default DMA offset to 0080H.

P_CLI creates the new process with a P_CREATE 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 P_TERMCPM or P_TERM.

You can also terminate a process by typing a single CTRL-C during console input. See C_MODE in Section 6 for the 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.

Note: Additional UDA space is allocated for 8087 processing only if the process is initialized by P_CLI. Other processes (such as RSPs) that require 8087 processing and do not use P_CLI must allocate this additional UDA space themselves.

3.1.1 Shared Code

Concurrent allows processes to share program code. This capability avoids unnecessary program loading of a code segment already in memory and conserves memory space since multiple copies of the same program code do not have to occupy different memory space.

When loading "sharable" program code, Concurrent allocates the code group separately from the rest of the program, and maintains this code group in memory even after the program has terminated. Subsequent loading of the same program does not load the code group, but uses the existing one instead. Obviously, programs written with separate code and data can take advantage of this feature.

Concurrent maintains a shared code group in memory until a memory request or a reset drive forces its release. Concurrrent maintains shared code groups in memory in **Least Recently Used** (LRU) order on the Shared Code List. If a memory request is made that cannot be satisfied, the list is drained, one at a time, until the memory request is satisfied, or the Shared Code List is emptied. If a drive is reset, Concurrent purges all code groups loaded from that drive.

A shared code program is flagged by the value 09H in G-Type field of the Code Group Descriptor in the CMD file header (see Section 3.2). You can set this field with the CHSET utility (see the User's Guide). Note that programs using the 8080 memory model cannot share code.

3.1.2 8087 Support

Concurrent provides optional 8087 support for systems that use the 8087 coprocessor. This support is indicated by the **Program Flag**, byte 127 (07FH), of the CMD file header.

Setting bit 6 (bit 0 is least significant bit) of the Program Flag indicates optional 8087 support, which means that if the 8087 is present, the program will use it; otherwise, the program will emulate it.

If bit 5 of the Program Flag is set, it indicates that the 8087 must be present in order for the program to run. If no 8087 is present and bit 5 of the Program Flag is set, the system returns an error when it tries to load the program. You can use the CHSET utility to set the program's header record for optional or required 8087 support.

If you use P_CLI to initiate and execute a process, Concurrent allocates an extra 96 bytes to the UDA for 8087 support. If you require 8087 support and do not use P_CLI, you must specifically allocate this additional 96 bytes to the UDA, turn on the 8087 flag in the PD, and initialize the CW and SW fields in the 8087 UDA extension (see description of these fields in Section 6 under P_CREATE).

3.1.3 8087 Exception Handling

Although Concurrent provides its own 8087 exception handling routine, you may want to write your own. Appendix B includes instructions and information required to write an 8087 exception handler, with a sample listing of such a routine.

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. Figure 3-1 shows the format of the header record.

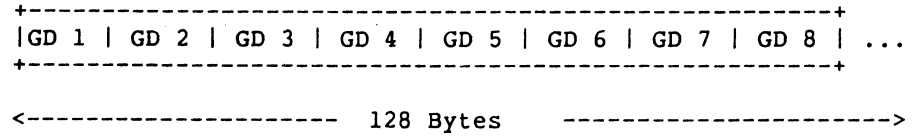


Figure 3-1. CMD File Header Format

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.

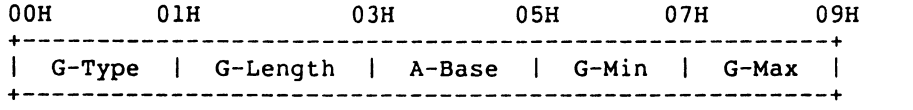


Figure 3-2. Group Descriptor Format

G_Type determines the Group Descriptor type. The valid Group Descriptors have a G_Type in the range 1 through 9, 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 Descriptor Types

G_Type	Group Type
01H	Code Group (non-shared)
02H	Data Group
03H	Extra Group
04H	Stack Group
05H	Auxiliary Group #1
06H	Auxiliary Group #2
07H	Auxiliary Group #3
08H	Auxiliary Group #4
09H	Code Group (shared)

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. For example, given a G_length of 080H, the size of the group is 0800H (2048 decimal) bytes.
A_Base	defines the base paragraph address for a nonrelocatable group.
G_Min	defines the minimum size of the memory area to allocate to the group.
G_Max	defines the 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 Base Page contains default values and locations initialized by P_CLI and P_LOAD 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.

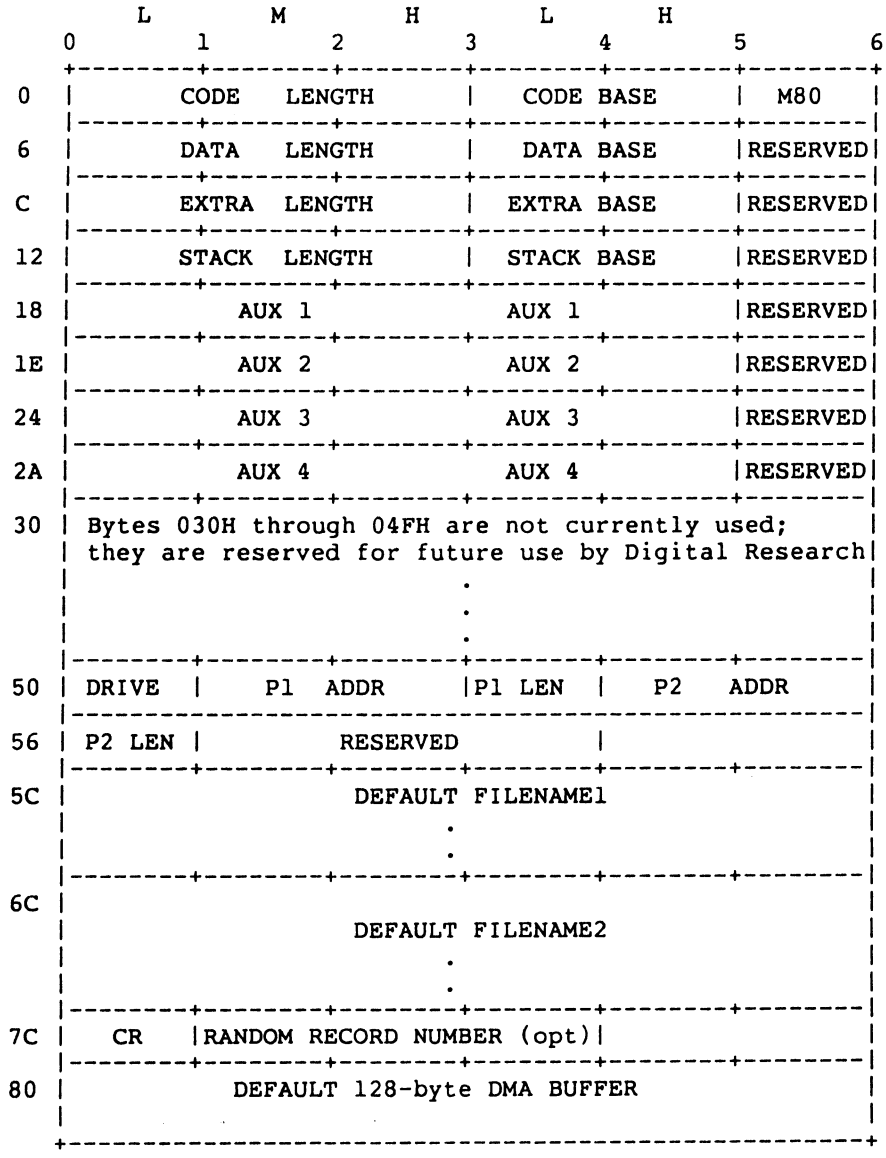


Figure 3-3. Base Page Values

Table 3-3 lists the fields in the Base Page.

Table 3-3. Base Page Fields

Field	Definition
M80	The M80 byte is a flag indicating whether the 8080 Memory Model was used during load. The values of the flag are: 1 = 8080 Model 0 = not 8080 Model If the 8080 Model is used, the code length never exceeds 0FFFFH.
AUX 1-4	Designate 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	Length is stored using the Intel convention: low, middle, and high bytes.
BASE	Refers to the paragraph address of the beginning of the segment.
DRIVE	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.
P1 ADDR	Contains the address of the password field of the first command tail operand in the default DMA buffer at 0080H. P_CLI sets this field to 0 if no password is specified.
P1 LEN	Contains the length of the password field for the first command tail operand. P_CLI sets this field to 0 if no password is specified.
P2 ADDR	Contains the address of the password field of the second command tail operand in the default DMA buffer at 0080H. P_CLI sets this field to 0 if no password is specified.
P2 LEN	Contains the length of the password field for the second command tail operand. P_CLI sets this field to 0 if no password is specified.

Table 3-3. (Cont'd)

Field	Definition
FILENAME1	Initialized by P_CLI for a transient program from the first command tail operand of the command line.
FILENAME2	Initialized by P_CLI 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.
CR	Contains the current record position used in sequential file operations with the FCB at 05CH.
RANDOM RECORD NUMBER	The optional Random Record Number is an extension of the FCB at 05CH, used in random record processing.
DMA BUFFER	The Default DMA buffer contains the command tail when P_CLI loads a transient program.

3.4 Parent/Child Process Relationships

Under Concurrent when one process (the parent) creates another process (the child), 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.

3.5 Direct Video Mapping

Processes which bypass Concurrent's Character I/O system calls and use a video map or screen buffer directly cannot be monitored, and continue to put characters on the screen even when running in the background. Consequently, any screen displayed by the program in the foreground console is interspersed with characters displayed by the program in the background using direct video map I/O.

To avoid the problems created by using direct video I/O, set bit 3 of the Program Flag to tell Concurrent that the process is to be put in suspend mode whenever it is running in the background and may continue running only when switched to the foreground. You can use the CHSET utility (see the User's Guide) to set bit 3 of the Program Flag.

Note that by-passing Concurrent's Character I/O system calls negates the concurrency of a process, because Concurrent suspends it from running (if bit 3 of Program Flag is set) unless it is running in the foreground.

End of Section 3

TRANSIENT PROGRAM MEMORY MODELS

When Concurrent loads a program, the initial values of the segment registers, the instruction pointer, and the stack pointer are determined by the memory model indicated in the CMD file header record.

There are three transient program models, the 8080 model, the Small Model, and the Compact Model, summarized in Table 4-1.

Table 4-1. Transient Program 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 containing 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 program during execution so that multiple segments in the code group can be addressed.

The **Small** Model is similar to that defined by Intel, consisting 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 a 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 program must manage its own segment registers during execution in order to address all code and data areas.

The three memory models differ primarily in how Concurrent initializes the segment registers when it loads a program. P_LOAD determines which memory model to use by examining the program group usage, as described in the following sections.

For all three memory models, Concurrent initializes an internal 96-byte stack. The first two words of this stack are reserved for the double word return for termination by a RETF (Far return) instruction. Figure 4-1 shows the initial program stack for all three memory models.

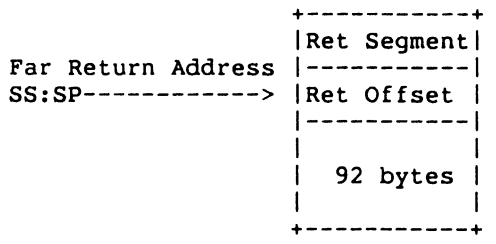


Figure 4-1. Initial Program Stack

The transient program can terminate by using P_TERMCPM or P_TERM, or by executing a RETF (Far Return) instruction when the SS and SP still point to the initial program stack.

4.1 The 8080 Memory Model

P_LOAD assumes the 8080 Model when the transient program contains only a code group. The intermixed code and data areas are indistinguishable. In this case, P_CLI (Command Line Interpreter) initializes the CS, DS, and ES registers to the beginning of the code group and sets the SS and SP registers to a 96-byte initial stack area that it allocates.

Note: P_CLI initializes the stack so that if the process executes a Far Return instruction, it terminates. P_CLI sets the Instruction Pointer (IP) Register to 100H, thus allowing Base Page values at the beginning of the code group. Following program load, the 8080 Model appears as shown in Figure 4-2.

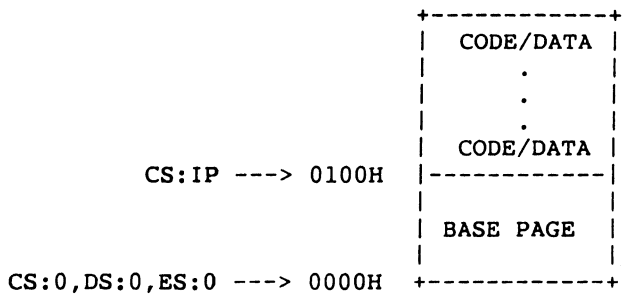


Figure 4-2. 8080 Memory Model

The following RASM-86 code fragment shows how to define an 8080 Model transient program.

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

4.2 The Small Memory Model

P_LOAD assumes the Small Model when the transient program contains both a code and data group. (In RASM-86, all code is generated following a CSEG directive. Data is defined following a DSEG directive, with the origin of the Data Segment independent of the Code Segment.)

In this model, P_CLI sets the CS register to the beginning of the code group, the IP to 0000H, the DS and ES registers to the beginning of the data group, and the SS and SP registers to a 96-byte initial stack area that it initializes. Following program load, the Small Model appears as shown in Figure 4-3.

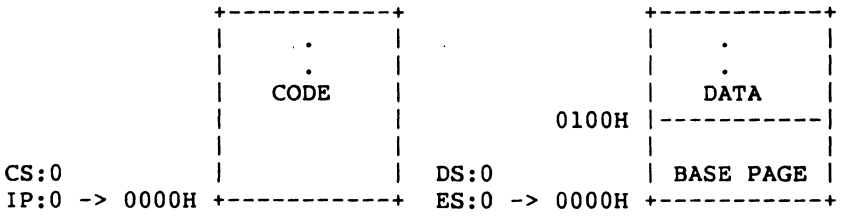


Figure 4-3. Small Memory Model

The machine code begins at CS+0000H, the Base Page values begin at DS+0000H, and the data area starts at DS+0100H.

The following RASM-86 code fragment shows how to define a Small Model transient program.

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

4.3 The Compact Memory Model

P_LOAD assumes the Compact Model when code and data groups are present, along with one or more of the remaining stack, extra, or auxiliary groups. In this case, P_CLI sets the CS, DS, and ES registers to the base addresses of their respective areas, with the IP set to 0000H, and the SS and SP registers set to a 96-byte stack area it allocates.

Figure 4-4 shows the initial configuration of the segments in the Compact Model. The values of the various segment registers can be changed during execution by loading from the initial values placed in Base Page. This allows access to the entire memory space.

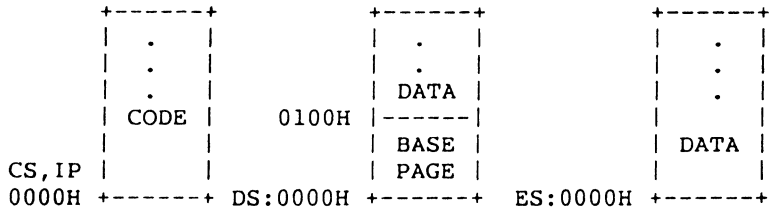


Figure 4-4. Compact Memory Model

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

Although it appears that the SS and SP registers should be set to address the stack group, there are two contradictions. First, the transient program might be using the stack group as a data area. In that case, the stack values set by P_CLI so a far return can terminate a transient program could overwrite data in the stack area. Second, the SS register would logically be set to the base of the group, while the SP would be set to the offset of the end of the group. However, if the stack group exceeds 64K, the address range from the base to the end of the group exceeds a 16-bit offset value.

The following RASM-86 code fragment shows how to define a Compact Model transient program.

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

End of Section 4

Resident System Process Generation

5.1 Introduction to RSPs

Resident System Processes are programs that become part of Concurrent during system generation. GENCCPM searches the directory for all files with the filetype RSP and prompts you to choose whether to include them in the system file, CCPM.SYS. You create an RSP file by generating a CMD file and then renaming it with an RSP filetype.

RSPs can be useful in several ways: to create a turnkey system, autoloading programs when Concurrent is booted; to build customized user interfaces or shells at the consoles, for monitoring hardware not supported in the XIOS; and to avoid disk loading time for frequently-used commands.

Appendix A includes the source code for the ECHO RSP. Study this listing carefully while reading this section. The discussion of P_CREATE in Section 6 is also helpful in understanding RSPs.

5.2 RSP Memory Models

RSPs have two memory models that are similar to the 8080 Model and the Small Model for transient programs. However, there are several important distinctions between transient program and RSP memory models.

The RSP has no equivalent of the Base Page in a transient program's Data Segment. The RSP is responsible for its own Process Descriptor (PD) and User Data Area (UDA). The RSP must also allocate an additional 96 bytes at the end of the User Data Area if 8087 processing is required. Concurrent automatically creates and initializes these data structures for transient programs at load time. RSPs, on the other hand, must initialize these structures within their own Data Segments (See P_CLI and P_CREATE in Section 6 for PD and UDA descriptions).

Although there is no Base Page in an RSP, there is an RSP header that must exist at offset 00H of the Data Segment. In the 8080 Model, this implies that the RSP header is in the Code Segment. Section 5.4 describes the RSP header and its associated data structures.

5.2.1 8080 Model RSP

The 8080 Model consists of mixed code and data. When Concurrent gives control of the CPU to an 8080 Model RSP, it initializes the Code, Data, Extra and Stack Segment registers to the same value.

GENCCPM assumes the 8080 Model if the RSP's CMD File Header Record has a single Code Group Descriptor and no other Group Descriptors (refer to Section 3.3). When discussing an 8080 Model RSP, any reference to the Data Segment also refers to the Code Segment.

5.2.2 Small Model RSP

The Small Model RSP implies separate Code and Data Segments. Before Concurrent gives control of the CPU to a Small Model RSP, it initializes the Data, Extra and Stack Segment Registers to the Data Segment address, while the Code Segment register is initialized to the Code Segment address.

There is no guarantee where GENCCPM will place the Code Segment in memory relative to the Data Segment. The CMD Header Record for a Small Model RSP must have both Data and Code Group Descriptors.

Figure 5-1 shows the 8080 and Small Memory model RSPs.

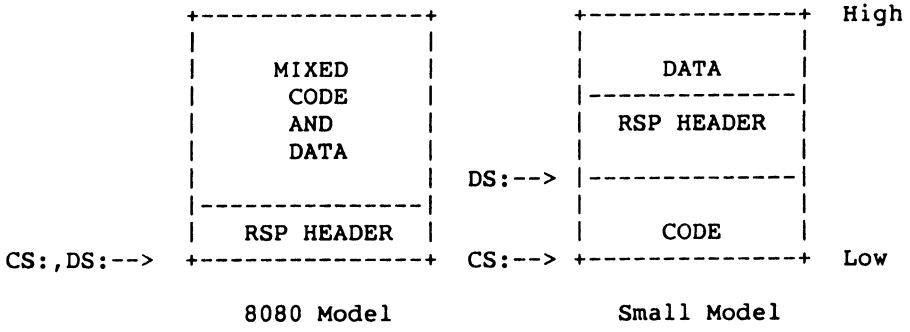


Figure 5-1. 8080 and Small Model RSPs

Note: Concurrent does not support compact model RSPs. Extra and Stack Segments must be part of the Data Segment.

5.3 Multiple Copies of RSPs

GENCCPM can make up to 255 copies of an RSP, with each copy generating a separate process. The number of copies made by GENCCPM can be fixed, or dependent on a byte value in the System Data Area. To determine the number of copies to make, GENCCPM examines the RSP Header. Figure 5-2 shows the RSP Header format.

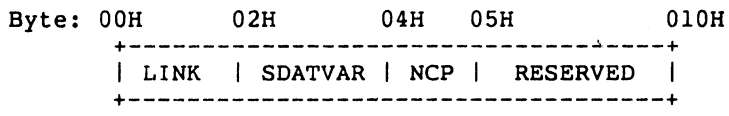


Figure 5-2. RSP Header Format

If the SDATVAR field is nonzero, GENCCPM uses it as an offset of a byte value in the System Data Area, which contains the number of copies to generate. The offset should indicate a value you set during system generation. The TMP RSP uses this feature by placing the offset of the NVCNS (Number of Virtual Consoles) field into the SDATVAR field. This way, a TMP is generated for each System Console you specify.

If SDATVAR is 0 then the NCP byte is used as the number or extra copies to make. If both of these fields are 0 then GENCCPM makes no extra copies. The ECHO RSP is an example of the latter.

If GENCCPM determines the number of copies is greater than 0, it gives each copy a unique copy number by placing the number in the NCP field and appending the ASCII equivalent to the end of the Process Descriptor NAME field of each copy. If there is not enough space for the number in the PD NAME, part of the PD NAME is over written.

For example, with the TMP RSP, GENCCPM makes the specified number of copies and changes the NAME field in each copy to be TMP0, TMP1, TMP2,..., and sets the NCP field to 0, 1, 2, ..., respectively.

5.3.1 8080 Model

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

5.3.2 Small Model

When GENCCPM makes copies of a Small Model RSP, it copies both the Code and Data Groups of the RSP, if the MEM field of the Process Descriptor is 0. See P_CREATE in Section 6 for a description of the Process Descriptor format. GENCCPM sets the UDA fields CS to the Code Segment of the RSP and DS, ES and SS to the Data Segment of the RSP.

5.3.3 Small Model with Shared Code

If a Small Model RSP has a nonzero MEM field in its Process Descriptor, the Code Segment is assumed to be reentrant. When GENCCPM makes copies of this type of RSP, it copies only the Data Group. GENCCPM sets the UDA CS field for each copy to the paragraph address of the one Code Segment for the RSP's, but sets the DS, ES, and SS, in each copied Data Segment to the paragraph address of the Data Segment for that particular copy.

5.4 Creating and Initializing an RSP

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

The first step of the loop reads a message from the RSP Command Queue. The process that writes the message to the RSP Command Queue activates the associated RSP. After the RSP returns from the Q_READ call, it obtains the system resources it needs, such as the calling process's console.

Typically, P_CLI assigns the RSP process the console process after successfully writing the command tail to the RSP Queue. This is only true if the RSP Process Descriptor name matches the RSP Command Queue name. See P_CLI in Section 6 for more information.

When the RSP completes its activities for the given command, it releases any system resources it has acquired, including the console, and restarts the loop by reading from its RSP Command Queue.

A Command RSP is a single process and is a serially reusable resource; in other words, the RSP acts on one message at a time. When several processes attempt to invoke a single Command RSP, they must wait. See Q_READ, Q_CREAD, Q_WRITE and Q_CWRITE in Section 6 for further details.

Note: It is certainly possible to create RSPs that are invoked differently.

Figure 5-3 shows the RSP Command Queue Message format.

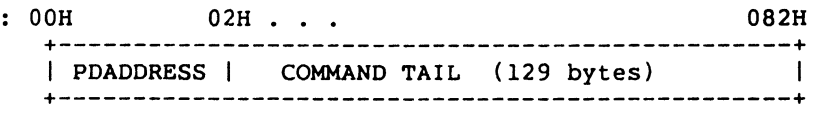


Figure 5-3. RSP Command Queue Message

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

When you enter a command at a console, the TMP performs a P_CLI call which attempts to open a queue that has the RSP Flag on, and has the same name as the command sent to the CLI. If the Q_OPEN is successful, P_CLI attempts to assign the calling process's console to a process with the same name as the command.

P_CLI then creates an RSP Command Queue Message with the command tail sent to the CLI from the TMP, and writes it to the RSP Command Queue. A transient program can use a Command RSP in the same manner by writing directly to the appropriate RSP Command Queue. An advantage of using P_CLI is that it looks for an RSP first and only searches on disk for a CMD file if the the RSP is not found.

When an RSP reads a RSP Command Queue Message, it often needs information about the calling process, such as which console, list device, drive, or user number to use. If a P_CLI call invokes an RSP, the RSP is assigned the calling process's console, but if the RSP Command Queue is written to directly, the calling process might or might not assign its console to the RSP.

A Command RSP can use the PD address in the Command RSP Message to find out what the default devices of the calling process are. The RSP should release any resources it assigns to itself when it is finished.

The RSP Header begins at offset 0 from the beginning of the RSP Data Segment. Note that in the 8080 Model, the RSP Header is also in the Code Segment. After the RSP Header is a Process Descriptor starting at offset 010H. A User Data Area and a stack must also be within the Data Segment, with the UDA placed at a paragraph boundary relative to the beginning of the Data Segment.

If system calls assuming a default DMA buffer are used, a 128-byte DMA Buffer must also exist. The DMA OFFSET field in the User Data Area should be set to the address of the DMA buffer. When Concurrent creates the process, the DMA SEGMENT field is initialized to the same value as the DS register. The DMA SEGMENT and OFFSET can also be set by calling F_DMASEG and F_DMAOFF once the RSP is running.

Figure 5-4 shows the beginning of the RSP Data Segment.

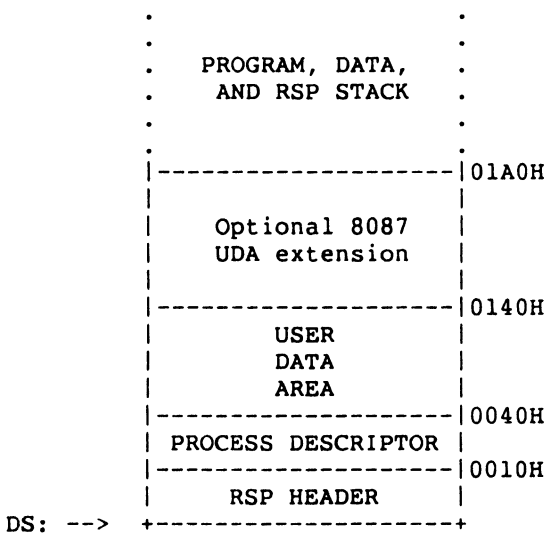


Figure 5-4. RSP Data Segment

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

5.4.1 The RSP Header

As discussed in Section 5.2, the number of RSP copies made depends on the values of the SDATVAR and NCP fields in the RSP Header. If no copies are desired, these fields must be zero. As a convenience, when Concurrent creates the RSP process, the LINK field in the RSP Header is set to the paragraph address of the System Data Area. The System Data Area can be obtained by an RSP or transient program with the S_SYSDAT call.

5.4.2 The RSP Process Descriptor

The RSP Process Descriptor should be initialed to zeros, except for the PRIORITY, FLAGS, NAME, and UDA SEGMENT fields. The PRIORITY field is usually initialized to 190. This is higher than transient programs and TMPs (200 and 198 respectively), but lower than the INIT process, which has a priority of 1. The description of P_PRIORITY in Section 6 contains more information about system priority assignments.

Starting an RSP at a priority of 190 ensures the RSP is able to create and open an RSP Command Queue before it can be invoked through a TMP. RSPs such as ECHO usually set their priority to 200 after creating and opening their RSP Command queue and before attempting to read from the queue.

Note: There are no guarantees about the order in which Concurrent creates the RSPs. If one RSP must run before another, it must have a higher priority. Such is the case when one RSP uses a resource created by a second RSP; the second must run (at least during initialization) with a priority higher than the first.

The Process Descriptor SYS and KEEP Flags can be initialized in the RSP Data Segment. The SYS Flag allows a process to read and write to and from restricted system queues. This is discussed below with regard to RSP Command Queues. The KEEP flag signals Concurrent that the process cannot be terminated. KEEP is necessary if an RSP is not to be terminated when you type CTRL-C on a console being used by the RSP. The 8087 flag tells Concurrent that a process is actively using the 8087 processor.

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

The UDA field in the Process Descriptor must be the offset (in paragraphs) of the UDA relative to the RSP data segment. GENCCPM restores the UDA field in the Process Descriptor to the actual UDA paragraph address when the system is loaded.

If the PD field name is not the same as the Command Queue, the CLI does not assign the console to the RSP.

5.4.3 The RSP User Data Area

The User Data Area must have its SP field set to the offset of a three-word IRET structure, in the RSP's Data Segment. The offset is relative to the beginning of the Data Segment. The first of the three words is the offset of the code entry point for the RSP, relative to the beginning of the RSP Code Segment. Concurrent executes an IRET instruction to start the RSP using these three words for the IP, CS and Flag registers respectively.

The CS value on the stack is initialized to be the CS field of the UDA, while the Flag value is set to 0200H (interrupts on). The RSP stack must come immediately before these three words. The initial values of the AX, BX, CX, DX, DI, SI, and BP registers are taken from the appropriate fields in the UDA.

The DMA OFFSET field should be set to the offset of the DMA buffer in the RSP's Data Segment. Except for the SP and DMA OFFSET fields, and possibly the AX, BX, CX, DX, DI, SI, and BP fields, the remainder of the UDA fields should be initialized to 0. The CS, DS, ES, and SS fields are set by GENCCPM as discussed in Section 5.3.

If you include the 8087 extension in the UDA, you must initialize the CW field (Control Word) to 03FFH and the SW (Status Word) field to 0 before system generation.

5.4.4 The RSP Stack

The RSP must reserve space for its stack, which is assumed to lie within the RSP's Data Segment. The RSP stack must be large enough to accommodate what the RSP code needs, plus four levels (eight bytes) to handle possible hardware interrupts. We highly recommend that you reserve more than four extra levels of stack.

The SP field in the RSP's UDA points to the top of this stack; the top contains the three-word IRET instruction discussed above.

5.4.5 The RSP Command Queue

The RSP's Command Queue contains information that determines when it begins execution, and to which console it is attached. If an RSP is to be accessible from a console via the TMP, the Command Queue name must be in upper-case. The FLAGS field in the RSP Command Queue Descriptor must have the RSP bit on. If this flag is not on, P_CLI does not write a message to the RSP Command Queue, and instead attempts to load a transient program. The KEEP flag should be set on to protect the RSP QUEUE from inadvertently using a Q_DELETE call.

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

The Queue Descriptor of the RSP Command Queue must have have a message length 131 bytes. The number of messages is usually 1. If the Queue Descriptor is within 64K bytes of the beginning of the System Data Area, buffer space for the Queue Descriptor must be allocated in the RSP. The BUFFER field in the Queue Descriptor must be the offset of this buffer, relative to the beginning of the RSP's Data Segment. The buffer size is the message length times the number of messages, usually 131 bytes.

Note: The queue buffer should be before the Queue Descriptor within the RSP Data Segment.

An RSP can certainly create other queues besides the RSP Command Queue used with Command RSPs. However, any queue an RSP creates that lies within 64K of the System Data Area must have a buffer area pointed to by the BUFFER field in its Queue Descriptor. To be safe, the buffer should come before the Queue Descriptor in the RSP's Data Segment.

It is assumed the BUFFER field points to a buffer that is also within 64K of the System Data Area. If the Queue Descriptor is farther than 64K from then System Data Area, Concurrent uses buffer space in the System Data Area. Refer to Q_MAKE in Section 6 for further details.

In order to open the RSP Command Queue and subsequently read from it, a Queue Parameter Block and its associated buffer must be allocated in the RSP's Data Segment. These structures are treated just as in a transient process.

Note: For any queues created by an RSP, the Queue Buffer areas associated with the Queue Descriptor and the Queue Parameter Block are separate, distinct areas of storage.

5.4.6 Multiple Processes within an RSP

An RSP can create child processes by calling P_CREATE. Note that if the Process Descriptor of the process being created is within 64K bytes of the beginning of the System Data Area, Concurrent uses the PD structure directly. Otherwise it copies the PD structure into the PD table in the System Data Area.

5.5 Developing and Debugging an RSP

The first RSP you attempt should be very simple, on the order of the ECHO RSP. New RSPs should be developed and debugged as if they were transient processes, such as Concurrent's CMD utilities, then converted into RSPs.

An RSP debugging session should proceed like an XIOS debugging session: first load CP/M-86, then invoke SID-86™, and then bring up Concurrent. The System Guide provides more information about running Concurrent under CP/M-86.

After reading in the CCPM.SYS file under SID-86, find the RSPSEG field of the System Data Segment (SYSDAT). The paragraph address of the SYSDAT is found in the A_BASE field of the Data Group Descriptor in the CCPM.SYS command file header. The RSPSEG field contains the paragraph address of the Data Segment of the first RSP in a linked list of the RSPs included by GENCCPM.

See S_SYSDAT in Section 6 for details of SYSDAT.

Using SID-86's Display Memory (D) command to show memory at the segment RSPSEG, you can identify the name of the first RSP in the RSP's Process Descriptor. The LINK field in the RSP Header, which is the first word in the RSPSEG segment, is the paragraph value of the next RSP's Data Segment. A zero in the LINK field means the end of the list of RSPs.

Note that linkage information is lost once Concurrent is initialized. The LINK field of the RSP Header contains the System Data Segment once an RSP begins execution.

Once you locate the RSP to be debugged, the initial code entry point can also be found. As discussed previously, the SP field in the RSP's UDA is the offset from the beginning of the RSP's Data Segment of the three-word IRET structure. The first word of the IRET structure contains the initial value of the IP register when Concurrent creates the RSP process. The initial value of the CS register is in the CS field also in the RSP's UDA. Once this is done, you can set break points in the RSP, similar to setting break points in XIOS system calls.

End of Section 5

Concurrent System Calls

This section summarizes the Concurrent DOS 86 system calls in tabular form. It is intended both as an introduction to the calls and as a reference for use when programming. You should be familiar with the material in Sections 1 through 5 before proceeding.

Note: The system calls described in this section are native to Concurrent. Section 7 describes the DOS system calls that Concurrent emulates. It is strongly recommended that you do not mix Concurrent and DOS system calls in the same program; code purely in one or the other.

6.1 Reference Tables

Table 6-1 describes the functional categories of Concurrent system calls and their general uses.

Table 6-2 lists the system calls in each category and serves as a quick reference to find the call you need while programming.

Table 6-3 is a summary of the system calls in alphabetical order (by mnemonics) along with the parameters you must pass when making the call, and the values returned by the call.

Table 6-4 lists the system calls numerically by function number.

Table 6-5 lists the error codes returned in register CX.

Table 6-6 is an index of the page numbers and figure titles for commonly used data structures.

Table 6-1. System Call Functional Categories

Category	Use
A_ Auxiliary Device I/O System Calls	The Auxiliary Device I/O system calls support I/O operations for auxiliary devices.
C_ Console System Calls	The Console system calls handle I/O operations for virtual consoles on a character, string, and line basis, attach and detach consoles from processes, and return or change the number corresponding to the default virtual console.
DEV_ Device System Calls	The Device system calls deal with flags and polling in managing system resources.
DRV_ Disk Drive System Calls	The Disk Drive system calls manage Concurrent's logical drives.
F_ File-Access System Calls	The File-Access system calls include calls that operate on files within a directory, calls that operate on records within files, and other miscellaneous system calls related to file I/O.
L_ List Device System Calls	The List Device system calls write characters or strings to the default list device, attach and detach the default list device from calling processes, and return or change the number corresponding to the default list device.
M_ MP/M-86™ Memory Management System Calls	The M_ Memory Management system calls are included for compatibility with MP/M-86. These calls allocate and free memory segments according to the MP/M-86 segmentation algorithm.
MC_ CP/M-86™ Memory Management System Calls	The MC_ Memory Management system calls allocate and free memory segments according to the CP/M-86 segmentation algorithm.

Table 6-1. (Cont'd)

Category	Use
P_ Process/Program System Calls	The Process/Program system calls create and terminate processes, call other processes, and perform other operations on processes.
Q_ Queue Management System Calls	The Queue Management system calls create, delete, open, read from, and write to queues.
S_ System Information Calls	The System information calls return various types of systems data, such as version numbers and addresses.
T_ Time System Calls	The Time system calls set the system calendar and clock and return the time from them in hours and minutes or in hours, minutes, and seconds.

Table 6-2. Concurrent DOS 86 System Calls

Mnemonic	Definition
	Auxiliary Device I/O Calls
A_ATTACH	Attach default auxiliary device to calling process.
A_CATTACH	Conditionally attach default auxiliary device to calling process.
A_DETACH	Detach default auxiliary device from calling process.
A_GET	Return default auxiliary device of calling process.
A_READ	Read a character from the default auxiliary device.
A_READBLK	Read characters from the default auxiliary input device and write them to a buffer.
A_SET	Set default auxiliary device for calling process.
A_STATIN	Obtain input status of default auxiliary input device.
A_STATOUT	Obtain output status of default auxiliary output device.
A_WRITE	Write a character to the default auxiliary output device.
A_WRITEBLK	Write a character string to the default auxiliary output device.
	Console I/O Calls
C_ASSIGN	Assign default virtual console to another process.
C_ATTACH	Establish ownership of the default virtual console to the calling process; suspend process until console becomes available.
C_CATTACH	Conditionally establish ownership of the default virtual console by the calling process; return an error message if the device is unavailable.
C_DELIMIT	Set or return current String Output Delimiter; used with C_WRITESTR.
C_DETACH	Detach default virtual console from the calling process.
C_GET	Return the virtual console number of the calling process.
C_MODE	Set or return Console mode.
C_RAWIO	Perform Raw mode I/O with the default virtual console.
C_READ	Read a character from the default virtual console.
C_READSTR	Read an edited line from the default virtual console.

Table 6-2. (Cont'd)

Mnemonic	Definition
C_SET	Set or change the default virtual console for the calling process.
C_STAT	Obtain the input status of the default virtual console.
C_WRITE	Write a character to the default virtual console.
C_WRITEBLK	Write a specified number (block) of characters to the default virtual console.
C_WRITESTR	Write a string to the default virtual console until delimiter.
	Device Calls
DEV_POLL	Poll a noninterrupt-driven device.
DEV_SETFLAG	Set a system flag.
DEV_WAITFLAG	Wait for a system flag to be set before restoring the current process.
	Disk Drive Calls
DRV_ACCESS	Indicate access to specified drives.
DRV_ALLOCVEC	Get the address of the disk Allocation Vector.
DRV_ALLRESET	Reset all disk drives.
DRV_DPB	Return the segment and offset address of the Disk Parameter Block for the default disk of the calling process.
DRV_FLUSH	Write internal pending blocking/deblocking data buffers to disk.
DRV_FREE	Relinquish access to specified drives.
DRV_GET	Return the default drive of the calling process.
DRV_GETLABEL	Return the directory label data byte for the specified drive.
DRV_LOGINVEC	Return bit map of logged-in disk drives.
DRV_RESET	Reset the specified drives.
DRV_ROVEC	Return bit map vector of drives set to Read-Only.
DRV_SET	Set default drive of calling process.
DRV_SETLABEL	Create or update a directory label.
DRV_SETRO	Set the default drive to Read-Only.
DRV_SPACE	Return unallocated space on the specified drive.

Table 6-2. (Cont'd)

Mnemonic	Definition
	File System Calls
F_ATTRIB	Set file attributes.
F_CLOSE	Close file.
F_DELETE	Delete file.
F_DMAGET	Return segment and offset address of Direct Memory Address buffer.
F_DMAOFF	Set the Direct Memory Address offset address.
F_DMASEG	Set Direct Memory Address buffer segment address.
F_ERRMODE	Set the BDOS Error mode.
F_LOCK	Lock record within file opened in Unlocked mode.
F_MAKE	Create file.
F_MULTISEC	Set the BDOS Multisector Count.
F_OPEN	Open file for record access.
F_PARSE	Parse an ASCII string and initialize an FCB.
F_PASSWD	Set the default password.
F_RANDREC	Set the Random Record field in the FCB from the sequential record position.
F_READ	Read records sequentially.
F_READRAND	Read random records
F_RENAME	Rename file.
F_SETDATE	Set file time and date stamp.
F_SFIRST	Search for first matching directory FCB that matches the specified FCB.
F_SIZE	Return the size of a file.
F_SNEXT	Search for next matching directory FCB that matches the FCB specified in the F_SFIRST system call.
F_TIMEDATE	Return file's date and time stamps and password mode.
F_TRUNCATE	Truncate file to the specified Random Record Number.
F_UNLOCK	Remove record locks.
F_USERNUM	Set or return the default user number of the calling process.

Table 6-2. (Cont'd)

Mnemonic	Definition
F_WRITE	Write records sequentially.
F_WRITERAND	Write random records.
F_WRITEXFCB	Create or update file's XFCB.
F_WRITEZF	Write random records and zero-fill any previously unallocated data blocks.
	List Device Calls
L_ATTACH	Establish ownership of the default list device by the calling process; suspend the process until the device is available.
L_CATTACH	Conditionally establish ownership of the default list device by the calling process; return error code if the device is unavailable.
L_DETACH	Relinquish ownership of the default list device.
L_GET	Return the default list device number of the calling process.
L_SET	Change the default list device for the calling process.
L_WRITE	Write a character to the default list device.
L_WRITEBLK	Write the specified number of characters (block) to the default list device.
	MP/M-compatible Memory Allocation Calls
M_ALLOC	Allocate the memory segment between the sizes specified in the Memory Parameter Block to the calling process.
M_FREE	Free the specified memory segment.
	CP/M-compatible Memory Allocation Calls
MC_ABSALLOC	Allocate a specified amount of RAM at a specific address.
MC_ABSMAX	Allocate the maximum amount of RAM available at a specified address.
MC_ALLFREE	Free all memory owned by the calling process.
MC_ALLOC	Allocate a segment of RAM, as specified in the Memory Control Block, to the calling process.
MC_FREE	Free an area of RAM beginning at a specified address, and extending to the end of the previously-allocated memory area.
MC_MAX	Allocate the maximum amount of RAM available in the system.

Table 6-2. (Cont'd)

Mnemonic	Definition
	Process/Program Calls
P_ABORT	Terminate a process specified by name or Process Descriptor address.
P_CHAIN	Pass control to the program specified in the DMA buffer.
P_CLI	Interpret and execute the specified command line by calling Command Line Interpreter (CLI).
P_CREATE	Create a subprocess.
P_DELAY	Suspend the calling process for a specified number of system clock ticks.
P_DISPATCH	Force a dispatch operation; give up the CPU resource to the highest priority process ready to run.
P_LOAD	Load the specified CMD file in memory; return its Base Page segment address.
P_PDADR	Return the address of the Process Descriptor of the calling process.
P_PRIORITY	Set the priority of the calling process.
P_RPL	Invoke a system call from a Resident Procedure Library.
P_TERM	Terminate the calling process.
P_TERMCPM	Terminate calling process unconditionally, release all owned resources.
	Queue Management Calls
Q_CREAD	Conditionally read a message from a system queue; return error code if a message is not available.
Q_CWRITE	Conditionally write a message to a system queue; return an error code if space is not available.
Q_DELETE	Delete a system queue.
Q_MAKE	Create a system queue.
Q_OPEN	Open a system queue for subsequent queue operations.
Q_READ	Read a message from a system queue; suspend calling process until message is available.
Q_WRITE	Write a message to a system queue; suspend calling process until space becomes available.

Table 6-2. (Cont'd)

Mnemonic	Definition
	System Information Calls
S_BDOSVER	Return BDOS version number, CPU and operating system type.
S_BIOS	Call specified CP/M-86 BIOS character I/O routine.
S_OSVER	Return type and version number of Concurrent.
S_SERIAL	Return the Concurrent system serial number.
S_SYSDAT	Return address of the System Data Segment (SYSDAT)
	Time Calls
T_GET	Obtain the system calendar and clock, hours and minutes only.
T_SECONDS	Return current system date and time; hours, minutes, seconds.
T_SET	Set internal system calendar and clock to specified value.

Table 6-3. System Call Summary - By Mnemonic

Mnemonic	Parameters	Returned Values
A_ATTACH (A5H)	none	none
A_CATTACH (A7H)	none	AX = 0000 if attach, FFFF on failure
A_DETACH (A6H)	none	AX = 0000 if detach, FFFF on failure
		CX = Error Code
A_GET (A9H)	none	AL = Aux Dev #
A_READ (03H)	none	AL = char
A_READBLK (172H)	DX = .CHCB	AX = # of chars read
A_SET (A8H)	AL = Aux #	AX = 0000 if set, FFFF on failure
		CX = Error Code
A_STATIN (07H)	none	AL = FFH/00H
A_STATOUT (08H)	none	AL = FFH/00H
A_WRITE (04H)	DL = char	none
A_WRITEBLK (ADH)	DX = .CHCB	AX = # of chars written
C_ASSIGN (95H)	DX = .ACB	AX = Return Code
C_ATTACH (92H)	none	none
C_CATTACH (A2H)	none	AX = Return Code
C_DELIMIT (6EH)	DX = Out Delim	AL = Out Delim
C_DETACH (93H)	none	none
C_GET (99H)	none	AL = con #
C_MODE (6DH)	DX = Con Mode	none
	= FFFFH	AX = Con Mode
C_RAWIO (06H)	see def	see def
C_READ (01H)	none	AL = char
C_READSTR (0AH)	DX = .Buffer	see def
C_SET (94H)	DL = Console	none
C_STAT (0BH)	none	AL = 00/01
C_WRITE (02H)	DL = char	none
C_WRITEBLK (6FH)	DX = .CHCB	none
C_WRITESTR (09H)	DX = .Buffer	none
DEV_POLL (83H)	DL = Device	none
DEV_SETFLAG (85H)	DL = Flag	AX = Return Code
DEV_WAITFLAG (84H)	DL = Flag	AX = Return Code
DRV_ACCESS (26H)	DX = Drive Vect	none
DRV_ALLOCVEC (1BH)	none	AX = .Alloc
DRV_ALLRESET (0DH)	none	see def
DRV_DPB (1FH)	none	AX = .DPB
DRV_FLUSH (30H)	none	see def
DRV_FREE (27H)	DX = Drive Vect	none
DRV_GET (19H)	none	AL = Cur Drive #
DRV_GETLABEL (65H)	DX = Drive #	AL = Label Data Byte

Table 6-3. (Cont'd)

Mnemonic	Parameters	Returned Values
DRV_LOGINVEC (18H)	none	AX = Login Vect
DRV_RESET (25H)	DX = Drive Vect	AL = Error Code
DRV_ROVEC (1DH)	none	AX = R/O Vect
DRV_SET (0EH)	DL = Drive #	see def
DRV_SETLABEL (64H)	DX = .FCB	AL = Dir Code
DRV_SETRO (1CH)	none	see def
DRV_SPACE (2EH)	DL = Drive #	see def
F_ATTRIB (1EH)	DX = .FCB	see def
F_CLOSE (10H)	DX = .FCB	AL = Dir Code
F_DELETE (13H)	DX = .FCB	AL = Dir Code
F_DMAGET (34H)	none	AX = DMA Offset
F_DMAOFF (1AH)	DX = .DMA	none
F_DMASEG (33H)	DX = .DMA Seg	none
F_ERRMODE (2DH)	DL = Err Mode	none
F_LOCK (2AH)	DX = .FCB	AL = Error Code
F_MAKE (16H)	DX = .FCB	AL = Dir Code
F_MULTISEC (2CH)	DL = # of Recs	AL = Return Code
F_OPEN (0FH)	DX = .FCB	AL = Dir Code
F_PARSE (98H)	DX = .PFCB	see def
F_PASSWD (6AH)	DX = .Password	none
F_RANDREC (24H)	DX = .FCB	R0, R1, R2
F_READ (14H)	DX = .FCB	AL = Error Code
F_READRAND (21H)	DX = .FCB	AL = Error Code
F_RENAME (17H)	DX = .FCB	AL = Dir Code
F_SFIRST (11H)	DX = .FCB	AL = Dir Code
F_SIZE (23H)	DX = .FCB	R0, R1, R2 AL = Dir Code
F_SNEXT (12H)	none	AL = Dir Code
F_TIMEDATE (66H)	DX = .XFCB	AL = Dir Code
F_TRUNCATE (63H)	DX = .FCB	see def
F_UNLOCK (2BH)	DX = .FCB	AL = Error Code
F_USERNUM (20H)	DL = 0FFH (get) = User # (set)	AL = User # none
F_WRITE (15H)	DX = .FCB	AL = Error Code
F_WRITERAND (22H)	DX = .FCB	AL = Error Code
F_WRITEXFCB (67H)	DX = .XFCB	AL = Dir Code
F_WRITEZF (28H)	DX = .FCB	AL = Error Code
L_ATTACH (9EH)	none	none
L_CATTACH (A1H)	none	AX = Return Code
L_DETACH (9FH)	none	none

Table 6-3. (Cont'd)

Mnemonic	Parameters	Returned Values
L_GET (A4H)	none	AL = list #
L_SET (A0H)	DL = List #	none
L_WRITE (05H)	DL = char	none
L_WRITEBLK (70H)	DX = .CHCB	none
M_ALLOC (80H)	DX = .MPB	AX = Return Code
M_FREE (82H)	DX = .MPB	none
MC_ABSALLOC (38H)	DX = .MCB	see def
MC_ABSMAX (36H)	DX = .MCB	see def
MC_ALLFREE (3AH)	none	none
MC_ALLOC (37H)	DX = .MCB	see def
MC_FREE (39H)	DX = .MCB	see def
MC_MAX (35H)	DX = .MCB	see def
P_ABORT (9DH)	DX = .ABP	AX = Return Code
P_CHAIN (2FH)	see def	none
P_CLI (96H)	DX = .CLBUF	none
P_CREATE (90H)	DX = .PD	none
P_DELAY (8DH)	DX = #ticks	none
P_DISPATCH (8EH)	none	none
P_LOAD (3BH)	DX = .FCB	AX = BP Addr
P_PDADR (9CH)	none	AX = PD Addr
P_PRIORITY (91H)	DL = Priority	none
P_RPL (97H)	DX = .CPB	AX = result
Q_CREAD (8AH)	DX = .QPB	AX = Return Code
Q_CWRITE (8CH)	DX = .QPB	AX = Return Code
Q_DELETE (88H)	DX = .QPB	AX = Return Code
Q_MAKE (86H)	DX = .QD	none
Q_OPEN (87H)	DX = .QPB	AX = Return Code
Q_READ (89H)	DX = .QPB	none
Q_WRITE (8BH)	DX = .QPB	none
S_BDOSVER (0CH)	none	AX= Version#
S_BIOS (32H)	DX = .BD	AX = BIOS rtn
S_OSVER (A3H)	none	AX = Version #
S_SERIAL (6BH)	DX = .serial #	serial # set
S_SYSDAT (9AH)	none	AX = Sys Data Addr
T_GET (69H)	DX = .TOD	AL = seconds
T_SECONDS (9BH)	DX = .TOD	TOD filled in
T_SET (68H)	DX = .TOD	none

Note: Concurrent does not support System calls 3, 4, 7, and 8.

Table 6-3 uses the following conventions:

.	Address of
#	Number
ACB	Assign Control Block
APB	Abort Parameter Block
Addr	Address
BD	Bios Descriptor
BP	Base Page
Char	ASCII Character
CHCB	Character Control Block
CLBUF	Command Line Buffer
CPB	Call Parameter Block
Con	Console
Cur	Current
Delim	Delimiter
Dir	Directory
DMA	Direct Memory Address
FCB	File Control Block
MCB	Memory Control Block
MPB	Memory Parameter Block
Num	Number
Out	Output
PD	Process Descriptor
PFCB	Parse Filename Control Block
QD	Queue Descriptor
QPB	Queue Parameter Block
Rec	Record
Rtn	Return
Sys	System
TOD	Time of Day
Vect	Vector

Uppercase mnemonics refer to Data Structures; see the function definition. A "." before a Data Structure means the byte offset of the Data Structure. A Return Code is either 0 for success or 0FFFFH to indicate failure. When the Return Code in AX is 0FFFFH, CX is the Error Code (see Table 6-5). An error code returned in AL is specific to the BDOS system call that was made.

Table 6-4. System Call Summary by Function Number

Decimal	Hexadecimal	Mnemonic
0	0	P_TERMCPM
1	1	C_READ
2	2	C_WRITE
3	3	A_READ
4	4	A_WRITE
5	5	L_WRITE
6	6	C_RAWIO
7	7	A_STATIN
8	8	A_STATOUT
9	9	C_WRITESTR
10	A	C_READSTR
11	B	C_STAT
12	C	S_BDOSVER
13	D	DRV_ALLRESET
14	E	DRV_SET
15	F	F_OPEN
16	10	F_CLOSE
17	11	F_SFIRST
18	12	F_SNEXT
19	13	F_DELETE
20	14	F_READ
21	15	F_WRITE
22	16	F_MAKE
23	17	F_RENAME
24	18	DRV_LOGINVEC
25	19	DRV_GET
26	1A	F_DMAOFF
27	1B	DRV_ALLOCVEC
28	1C	DRV_SETRO
29	1D	DRV_ROVEC
30	1E	F_ATTRIB
31	1F	DRV_DPB
32	20	F_USERNUM
33	21	F_READRAND
34	22	F_WRITERAND
35	23	F_SIZE
36	24	F_RANDREC
37	25	DRV_RESET
38	26	DRV_ACCESS
39	27	DRV_FREE
40	28	F_WRITEZF

Table 6-4. (Cont'd)

Decimal	Hexadecimal	Mnemonic
42	2A	F_LOCK
43	2B	F_UNLOCK
44	2C	F_MULTISEC
45	2D	F_ERRMODE
46	2E	DRV_SPACE
47	2F	P_CHAIN
48	30	DRV_FLUSH
50	32	S_BIOS
51	33	F_DMASEG
52	34	F_DMAGET
53	35	MC_MAX
54	36	MC_ABSMAX
55	37	MC_ALLOC
56	38	MC_ABSALLOC
57	39	MC_FREE
58	3A	MC_ALLFREE
59	3B	P_LOAD
99	63	F_TRUNCATE
100	64	DRV_SETLABEL
101	65	DRV_GETLABEL
102	66	F_TIMEDATE
103	67	F_WRITEXFCB
104	68	T_SET
105	69	T_GET
106	6A	F_PASSWD
107	6B	S_SERIAL
109	6D	C_MODE
110	6E	C_DELIMIT
111	6F	C_WRITEBLK
112	70	L_WRITEBLK
116	74	F_SETDATE
128	80	M_ALLOC
129	81	M_ALLOC
130	82	M_FREE
131	83	DEV_POLL
132	84	DEV_WAITFLAG
133	85	DEV_SETFLAG
134	86	Q_MAKE
135	87	Q_OPEN
136	88	Q_DELETE
137	89	Q_READ

Table 6-4. (Cont'd)

Decimal	Hexadecimal	Mnemonic
138	8A	Q_CREAD
139	8B	Q_WRITE
140	8C	Q_CWRITE
141	8D	P_DELAY
142	8E	P_DISPATCH
143	8F	P_TERM
144	90	P_CREATE
145	91	P_PRIORITY
146	92	C_ATTACH
147	93	C_DETACH
148	94	C_SET
149	95	C_ASSIGN
150	96	P_CLI
151	97	P_RPL
152	98	F_PARSE
153	99	C_GET
154	9A	S_SYSDAT
155	9B	T_SECONDS
156	9C	P_PDADR
157	9D	P_ABORT
158	9E	L_ATTACH
159	9F	L_DETACH
160	A0	L_SET
161	A1	L_CATTACH
162	A2	C_CATTACH
163	A3	S_OSVER
164	A4	L_GET
165	A5	A_ATTACH
166	A6	A_DETACH
167	A7	A_CATTACH
168	A8	A_SET
169	A9	A_GET
172	AC	A_READBLK
173	AD	A_WRITEBLK

Table 6-5. Register CX Error Codes

Dec	Hex	Error Report
0	00H	No error
1	01H	System call not implemented
2	02H	Illegal system call number
3	03H	Cannot find memory
4	04H	Illegal flag number
5	05H	Flag overrun
6	06H	Flag underrun
7	07H	No unused Queue Descriptors
8	08H	No free queue buffer
9	09H	Cannot find queue
10	0AH	Queue in use
12	0CH	No free process descriptors
13	0DH	No queue access
14	0EH	Empty queue
15	0FH	Full queue
16	10H	CLI queue missing
17	11H	No 8087 in system
18	12H	No unused Memory Descriptors
19	13H	Illegal console number
20	14H	No Process Descriptor match
21	15H	No console match
22	16H	No CLI process
23	17H	Illegal disk number
24	18H	Illegal filename
25	19H	Illegal filetype
26	1AH	Character not ready
27	1BH	Illegal memory descriptor
28	1CH	Bad return from BDOS load
29	1DH	Bad return from BDOS read
30	1EH	Bad return from BDOS open
31	1FH	Null command
32	20H	Not owner of resource
33	21H	No CSEG in load file
34	22H	Process Descriptor exists on Thread Root
35	23H	Could not terminate process
36	24H	Cannot attach to process
37	25H	Illegal list device number
38	26H	Illegal password
40	28H	External termination occurred
41	29H	Fixup error upon load
42	2AH	Flag set ignored
43	2BH	Illegal auxiliary device number

Table 6-6. Data Structures Index

A_ATTACH

Attach Default Auxiliary Device to Calling Process

Entry Parameters:
Register CL: A5H (165)

A_ATTACH attaches the default auxiliary device to the calling process. If the auxiliary device is already owned by the calling process or if it is not owned by another process, A_ATTACH immediately returns with ownership established and verified. If another process owns the auxiliary device, the calling process relinquishes the CPU and waits until the auxiliary device becomes available.

The A_READ, A_READBLK, A_WRITE, and A_WRITEBLK calls internally invoke A_ATTACH to ensure the calling process owns the auxiliary device before attempting the appropriate read or write operation. If the process does not own the device, these Auxiliary Device I/O system calls use A_ATTACH to establish ownership.

The preferred method of performing auxiliary device I/O is to call A_ATTACH to establish device ownership before invoking the appropriate read or write system function.

A_CATTACH

Conditionally Attach Default Auxiliary Device To Calling Process

Entry Parameters:
Register CL: A7H (167)

Returned Values:
Register AX: 0000 on attach, FFFFH on failure
BX: Same as AX

A_CATTACH attaches the default auxiliary device to the calling process only if the device is currently available. If the auxiliary device is currently attached to another process, A_ATTACH returns a value of FFFFH to indicate the device could not be attached. A_CATTACH returns a value of 0000 to indicate that either the auxiliary device is already attached to the process, or that it was successful in attaching the device.

A_DETACH

Detach Default Auxiliary Device from Calling Process

Entry Parameters:
Register CL: A6H (166)

Returned Values:
Register AX: 0000 on detach, FFFFH on failure
BX: Same as AX
CX: Error code

A_DETACH detaches the default auxiliary device from the calling process. A_DETACH performs no action if the auxiliary device is not currently attached to the calling process.

Table 6-5 contains the list of error codes returned in register CX.

A_GET

Return the Calling Process's Default Auxiliary Device

Entry Parameters:
Register CL: A9H (169)

Returned Values:
Register AL: Auxiliary device number
BL: Same as AL

A_GET returns the default auxiliary device number of the calling process.

A_READ

Read a Character from the Default Auxiliary Input Device

Entry Parameters:
Register CL: 03H (3)

Returned Values:
Register AL: ASCII character
BL: Same as AL

A_READ reads the next 8-bit character from the logical auxiliary input device (AUXIN:) and returns it in register AL. Before reading the character, Concurrent internally calls A_ATTACH to ensure that the calling process owns its default auxiliary device (see A_ATTACH). A_READ does not return control to the calling process until it has read the character.

A_READBLK

Read Characters from the Default Auxiliary Input Device and Write Them to a Buffer

Entry Parameters:
Register CL: ACH (172)
DX: CHCB address

Returned Values:
Register AX: Number of characters read
BX: Same as AX

A_READBLK reads characters from the default auxiliary input device (AUXIN:) and writes them into the character buffer located by the **Character Control Block** (CHCB) addressed in DX. Concurrent calls A_ATTACH to ensure the calling process owns its default auxiliary device before performing the read operation (see A_ATTACH).

The format of the CHCB is as follows:

Bytes 0-1: Offset of character buffer
Bytes 2-3: Segment address of character buffer
Bytes 4-5: Length of character buffer

A_READBLK returns the number of characters actually read from the default auxiliary device in register AX. A_READBLK returns to the calling process when the status of AUXIN: indicates that the device is empty or the character buffer is full. A_READBLK does not return control to the calling process until at least one character has been read.

A_SET

Set the Calling Process's Default Auxiliary Device

Entry Parameters:
Register CL: A8H (168)
DL: Auxiliary device number

Returned Values:
Register AX: 0000 on set, FFFF on failure
BX: Same as AX
CX: Error code

A_SET sets the default auxiliary device for the calling process.

Table 6-5 contains the list of error codes returned in register CX.

A_STATIN

Obtain the Input Status of the Default Auxiliary Input Device

Entry Parameters:
Register CL: 07H (7)

Returned Values:
Register AL: FFH Character ready, 00 Not ready
BL: Same as AL

A_STATIN checks the input status of the AUXIN: device. If a character is ready for input from the auxiliary device, A_STATIN returns the value FFH in register AL. A_STATIN returns 00H if no input is ready.

A_STATOUT

Obtain the Output Status of the Default Auxiliary Output Device

Entry Parameters:
Register CL: 08H (8)

Returned Values:
Register AL: FFH Ready for output, 00 Not ready
BL: Same as AL

A_STATOUT checks the output status of the AUXOUT: device. If AUXOUT: is ready for output, A_STATOUT returns the value FFH in register AL. A_STATOUT returns 00H if the auxiliary device is not ready for output.

A_WRITE

Write a Character to the Default Auxiliary Output Device

Entry Parameters:
Register CL: 04H (4)
DL: ASCII character

A_WRITE writes the specified character to the default auxiliary device of the calling process. Before writing the character, A_WRITE calls A_ATTACH to ensure the calling process owns its default auxiliary device. If the process does not own the device, A_WRITE uses A_ATTACH to establish ownership (see A_ATTACH).

A_WRITEBLK

Send Specified Character String to
the Default Auxiliary Output Device

Entry Parameters:
Register CL: ADH (173)
DX: CHCB address

Returned Values:
Register AX: Number of characters written
BX: Same as AX

A_WRITEBLK writes the character string located by the **Character Control Block** (CHCB) addressed in register DX to the default auxiliary device. As with A_WRITE, Concurrent ensures the calling process owns the auxiliary device by calling A_ATTACH before attempting the write operation (see A_ATTACH).

The format of the CHCB is as follows:

Bytes 0-1: Offset of character string
Bytes 2-3: Segment address of character string
Bytes 4-5: Length of character string

A_WRITEBLK returns the number of characters written to the default auxiliary device in register AX. A_WRITEBLK returns to the calling process when the status of AUXOUT: indicates that the device is full or the character string has been written. A_WRITEBLK does not return control to the calling process until at least one character has been written.

C_ASSIGN

Assign Default Console Device To Another Process

Entry Parameters:
Register CL: 095H (149)
DX: ACB Address - Offset
DS: ACB Address - Segment

Returned Values:
Register AX: 0 if assign "OK", 0FFFFH on Failure
BX: Same as AX
CX: Error Code

C_ASSIGN directly assigns the specified console to a specified process, overriding the normal mechanism of the C_ATTACH and C_DETACH calls. C_ASSIGN returns an error code if a process other than the calling process owns the console. C_ASSIGN ignores other processes waiting to attach to the specified console, and they continue to wait until the current owner either calls C_DETACH, or terminates.

The calling process passes the address of an **Assign Control Block** (ACB). Figure 6-1 shows the Assign Control Block format. Table 6-7 lists the fields in the Assign Control Block. Table 6-5 contains the list of error codes returned in CX.

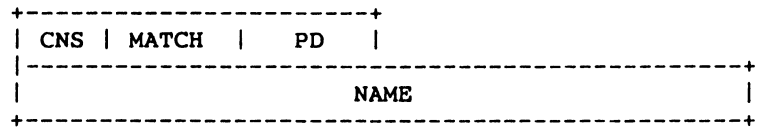


Figure 6-1. ACB - Assign Control Block

Table 6-7. ACB Field Definitions

Field	Definitions
CNS	Console to assign
MATCH	Boolean; if 0FFH, the process being assigned the console must have the CNS as its default console for a successful Assign. If 0H, no check is made.
PD	Process ID of the process being assigned the console. If this field is zero, a search is made of the Thread list for a process whose name is NAME. This field must be either zero or a valid Process ID. If this value is not a valid PD, an error occurs.
NAME	8-byte process name to search for. An error occurs if a process by this name does not exist.

C_ATTACH

Attach Default Console To Calling Process

Entry Parameters:
Register CL: 092H (146)

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

C_CATTACH

Conditionally Attach Default Console To Calling Process

Entry Parameters:
Register CL: 0A2H (162)
Returned Values:
Register AX: 0 if attach 'OK', 0FFFFH on failure
BX: Same as AX
CX: Error Code

C_CATTACH attaches the default console of the calling process only if the console is currently unattached. If the console is currently attached to another process, C_CATTACH returns a value of 0FFH indicating the console could not be attached. C_CATTACH returns a value of 0 to indicate that either the console is already attached to the process or that it was unattached and a successful attach operation was made.

Table 6-5 contains the list of error codes returned in CX.

C_DELIMIT

Set or Return Output Delimiter

Entry Parameters:
Register CL: 06EH (110)
DX: 0FFFFH (get) or
DL: Output Delimiter (set)

Returned Values:
Register AL: Output Delimiter, or no value if set
BL: Same as AL

C_DELIMIT can set or interrogate the current Output Delimiter. If register DX = 0FFFFH, then the current Output Delimiter is returned in register AL. Otherwise, C_DELIMIT sets the Output Delimiter to the value in register DL.

C_DELIMIT sets the string delimiter for C_WRITESTR. When a new process is created, the default delimiter value is set to a dollar sign, $. The default delimiter is not inherited from the parent process.

C_DETACH

Detach Default Console From Calling Process

Entry Parameters:
Register CL: 093H (147)

Returned Values:
Register AX: 0 if detach 'OK", 0FFFFH on failure
BX: Same as AX

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

C_GET

Return The Calling Process's Default Console

Entry Parameters:
Register CL: 099H (153)

Returned Values:
Register AL: Console number
BL: Same as AL

C_GET returns the default console number of the calling process.

C_MODE

Set or Return Console mode

Entry Parameters:
Register CL: 06DH (109)
DX: FFFFH (get) or Console mode (Set)

Returned Values:
Register AX: Console mode or (no value)
BX: Same as AX

C_MODE can set or interrogate the **Console Mode**, which is a 16-bit system parameter that determines the action of certain Console I/O functions. The Console Mode is set to zero when a new process created; it is not inherited from its parent.

If register DX = FFFFH, C_MODE returns the current Console Mode in register AX. Otherwise, C_MODE sets the Console Mode to the value contained in register DX.

The Console Mode definition is:

bit 0 = 1	CTRL-C only status for C_STAT.
bit 0 = 0	Normal status for C_STAT.
bit 1 = 1	Disable support for stop/start scroll (CTRL-S/CTRL-Q).
bit 1 = 0	Enable support for stop/start scroll.
bit 2 = 1	Raw console output mode. Disables tab expansion for C_WRITE, C_WRITESTR, and C_WRITEBLK. Also disables support for printer echo (CTRL-P).
bit 2 = 0	Normal console output mode.
bit 3 = 1	Disable CTRL-C program termination.
bit 3 = 0	Enable CTRL-C program termination.
bit 7 = 1	Disable CTRL-O console output byte bucket.
bit 7 = 0	Enable CTRL-O console output byte bucket
bit 10 = 1	Enable Escape as end-of-line character.
bit 10 = 0	Disable Escape as end-of-line character.

Note that the Console mode bits are numbered from right to left.

C_RAWIO

Perform Direct Console I/O With Default Console

Entry Parameters:
Register CL: 06H (6)
DL: 0FFH (Input/Status) or
0FEH (Status) or
0FDH (Input) or Character (Output)

Returned Values:
Register AL: (Input/Status)
= 0H (No Character)
= Character
(Status)
= 0H (No Character)
= 0FFH (Ready)
(Input)
= Character
(Output)
No return value
BL: Same as AL

C_RAWIO allows the calling process to do raw console I/O to its default console. Concurrent verifies that the calling process owns its default console before allowing any I/O.

In Raw mode, the CTRL-C, CTRL-P, CTRL-S, and CTRL-O characters are not acted on by the PIN (Physical Input Process) but are passed on to the calling process.

Note: If the virtual console is in CRTL-S mode, and the process that owns the virtual console then performs a C_RAWIO call, the CTRL-S state is reset.

A process calls C_RAWIO by passing one of three values shown in Table 6-8.

Table 6-8. C_RAWIO Calling Values

Value	Description
0FFH	Console input status command (if no character is ready, a 00H is returned, else the character is returned.)
0FEH	Console status command (on return, register AL contains 00H if no character is ready; otherwise it contains 0FFH.)
0FDH	Console input command (if no character is ready, the calling process waits until one is typed.) Input characters are not echoed to the screen.
ASCII character	If the parameter is less than 0FDH, C_RAWIO assumes register DL contains a valid ASCII character and sends it to the console.

C_READ

Read A Character From The Default Console

Entry Parameters:
Register CL: 01H (1)

Returned Values:
Register AL: Character
BL: Same as AL

C_READ reads a character from the default console of the calling process. Before attempting the read, Concurrent verifies the ownership of the console. If the calling process does not own the console, it relinquishes the CPU resource until the calling process can attach to the console. Typically, a process created through a P_CLI call owns its default console when it begins execution.

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

C_READ ignores the termination character (CTRL-C) if the calling process cannot terminate (refer to P_TERM). C_READ does not return until a character is typed on the console. Concurrent suspends the calling process until a character is ready.

C_READSTR

Read An Edited Line From The Default Console

Entry Parameters:
Register CL: 0AH (10)
DX: Console Buffer Address - Offset
DS: Console Buffer Address - Segment

C_READSTR reads characters from the calling process's default console and places them into the specified **Console Buffer**. Figure 6-2 shows the format of the Console Buffer, and Table 6-9 lists the Console Buffer field definitions.

C_READSTR performs line-editing system calls on the line as it is read from the console; it completes a line and returns upon receiving a terminator character (carriage return or line feed) from the console, or when the maximum number of characters is reached.

As with C_READ, C_READSTR echoes all graphic characters read from the console. Concurrent verifies that the calling process owns its default console before allowing I/O to begin.

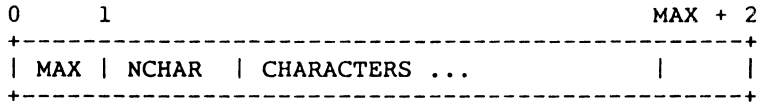


Figure 6-2. Console Buffer Format

Table 6-9. Console Buffer Field Definitions

Field	Definition
MAX	Maximum number of characters that can be read into the buffer. This value must be initialized before calling C_READSTR.
NCHAR	Actual number of characters read into the buffer as filled in by C_READSTR.
CHARACTERS	Actual characters read from the console as filled in by C_READSTR.

C_READSTR recognizes a number of special characters used in editing the input line, as well as a set of special characters that actually control the calling process.

Table 6-10. C_READSTR Line-editing Characters

Character	Function
CTRL S	Move the cursor one character to the left.
CTRL D	Move the cursor one character to the right.
CTRL A	Move the cursor one word to the left.
CTRL F	Move the cursor one word to the right.
CTRL Q	Move the cursor to the beginning of the line.
CTRL W	Move the cursor to the end of the line.
CTRL H	Delete the character to the left of the cursor.
CTRL G	Delete the character to the right of the cursor.
CTRL T	Delete the word to the right of the cursor.
CTRL Y	Delete entire line. If the line has been modified, it is saved in the commnd history buffer.
CTRL U	Delete to the beginning of the line.
CTRL K	Delete to the end of the line.
CTRL V	Toggle the insert mode.
CTRL \	Enter the next character literally.
CTRL E	Move up in the command history buffer. If the line has been modified, it is saved.
CTRL X	Move down in the command history buffer. If the line has been modified, it is saved.
CTRL J	Line feed; terminates the input line. C_READSTR does not echo a terminating character, nor does it place the character in the line buffer.
CTRL M	Carriage return; terminates the input line.

Table 6-10. (Cont'd)

Character	Function
CTRL R	Toggles the search mode on/off for the current line. After entering the line, the search mode returns to the default (off).
CTRL _	Toggles the default search mode on/off and sets the current line's mode to the new default. Initially, the default begins each line with the search mode off.
RUB/DEL	Same as CTRL-H.
BACKSPACE	Same as CTRL-H.

C_SET

Set The Calling Process's Default Console

Entry Parameters:
Register CL: 094H (148)
DL: Console Number

Returned Values:
AX: 0 if successful, 0FFFFH on failure
BX: Same as AX
CX: Error Code

C_SET changes the calling process's default console to the value specified. If the console number specified is not one supported by this particular implementation of Concurrent, C_SET returns an error code, and does not change the default console.

Table 6-5 contains the list of error codes returned in CX.

C_STAT

Obtain Status of the Default Console

Entry Parameters:
Register CL: 0BH (11)

Returned Values:
Register AL: 01H character ready, 00H not ready
BL: Same as AL

C_STAT checks to see if a character has been typed at the default console. If the calling process is not attached to its default console, C_STAT causes a dispatch to occur and returns 00H (the Not Ready condition).

C_STAT sets the console to the Nonraw mode, allowing recognition of special control characters such as the terminate character, CTRL-C. Use C_RAWIO to obtain console status in Raw mode.

Note: If C_MODE is used to set bit 0 in the console mode word, C_STAT only returns AL = 01H when a CTRL-C is typed on the default console.

C_WRITE

Write A Character To The Default Console

Entry Parameters:
Register CL: 02H (2)
DL: ASCII character

C_WRITE writes the specified character to the calling process's default console. As with C_READ, Concurrent verifies that the calling process owns its default console before performing the operation. On output, C_WRITE expands tabs in columns of eight characters.

C_WRITEBLK

Send Specified String to CONOUT:

Entry Parameters:
Register CL: 06FH (111)
DX: CHCB Address

C_WRITEBLK sends the character string located by the **Character Control Block** CHCB, addressed in register pair DX to the logical console, CONOUT:. If the Console mode is in the default state, C_WRITEBLK expands CTRL-I tab characters in columns of eight characters.

The CHCB format is:

bytes 0-1: Offset of character string
bytes 2-3: Segment of character string
bytes 4-5: Length of character string to print

C_WRITESTR

Print An ASCII String To The Default Console

Entry Parameters:
Register CL: 09H (9)
DX: STRING Address - Offset
DS: STRING Address - Segment

C_WRITESTR prints an ASCII string starting at the indicated string address and continuing until it reaches a dollar sign ($) character (024H; $ is the default string delimiter, and can be changed by with C_DELIMIT). C_WRITESTR writes this string to the calling process's default console.

Concurrent verifies that the calling process owns the console before writing the string. C_WRITESTR sets the console to a Nonraw state and expands tabs in columns of eight characters, as does C_WRITE.

Use C_WRITESTR whenever possible, rather than the single-character system calls. The CPU overhead involved in handling the first character is the same as that for a single-character system call, but subsequent characters require as little as one-fifth the CPU overhead.

DEV_POLL

Poll A Device

Entry Parameters:
Register CL: 083H (131)
DL: Device Number

Returned Values:
Register AX: 0 on success, 0FFFFH on failure
BX: Same as AX
CX: Error Code

DEV_POLL is used by the XIOS to poll noninterrupt-driven devices. It should be used whenever the XIOS is waiting for a noninterrupt event.

The calling process relinquishes the CPU and allows Concurrent to poll the device at every dispatch. The XIOS contains routines for each polling device number which are called through DEV_POLL, and they return whether the device is ready or not.

When the device is ready, DEV_POLL restores the calling process to the RUN state and returns. Upon return, the calling process knows the device is ready.

Table 6-5 contains the list of error codes returned in CX.

DEV_SETFLAG

Set A System Flag

Entry Parameters:
Register CL: 085H (133)
DL: Flag Number

Returned Values:
Register AX: 0 on success, 0FFFFH on failure
BX: Same as AX
CX: Error Code

DEV_SETFLAG is used by interrupt routines to notify Concurrent that a logical interrupt has occurred. A process waiting for this flag is placed back into the RUN state. If there are no processes waiting, then the next process to wait for this flag returns successfully without relinquishing the CPU. DEV_SETFLAG detects an error if the flag has already been set, and no process has done a DEV_WAITFLAG call to reset it.

Note: If a process waiting for a specific flag to be set is aborted, the next DEV_SETFLAG call is ignored and 0FFFFH is returned in AX.

Table 6-5 contains the list of error codes returned in CX.

DEV_WAITFLAG

Wait For A System Flag

Entry Parameters:
Register CL: 084H (132)
DL: Flag Number

Returned Values:
Register AX: 0 on success, 0FFFFH on failure
BX: Same as AX
CX: Error Code

DEV_WAITFLAG is used by a process to wait for an interrupt. The process relinquishes the CPU until an interrupt routine calls DEV_SETFLAG, which places the waiting process in the RUN state.

When DEV_WAITFLAG returns to the calling process, the interrupt has occurred, or an error has occurred. An error occurs when a process is already waiting for the flag. If the Flag was set before DEV_WAITFLAG was called, the routine returns successfully without relinquishing the CPU.

DEV_WAITFLAG is meant to be used by the XIOS. The mapping between types of interrupts and flag numbers is maintained in the XIOS, although Concurrent reserves flags 0, 1, 2, 3, and 4 for system use.

Table 6-5 contains the list of error codes returned in CX.

DRV_ACCESS

Access Specified Disk Drives

Entry Parameters:
Register CL: 026H (38)
DX: Drive Vector

Returned Values:
AL: Return Code
AH: Extended Error
BX: Same as AX

DRV_ACCESS inserts a special open file item into the system Lock List for each drive specified in the **Drive Vector**, which is passed in register DX. While the item exists in the Lock List, no other process can reset the drive. DRV_ACCESS inserts no items if insufficient free space exists in the Lock List to support all the new items, or if the number of items to be inserted puts the calling process over the Lock List open file maximum.

Figure 6-3 illustrates the format of the Drive Vector. The least significant bit corresponds to drive A, and the high-order bit corresponds to the sixteenth drive, labeled P.

Figure 6-3. Drive Vector Structure

If the BDOS is in the default Error mode (see F_ERRMODE), the file system displays a message at the console identifying the error and terminates the calling process. Otherwise, DRV_ACCESS returns to the calling process with register AL set to 0FFH and register AH set to one of the following hexadecimal values:

0AH - Open File Limit Exceeded
0BH - No Room in System Lock List

On successful calls, DRV_ACCESS returns with register AL set to 00H.

DRV_ALLOCVEC

Get Allocation Vector Address For the Calling Process's Default Disk

Entry Parameters:
Register CL: 01BH (27)

Returned Values:
Register AX: ALLOC Address - Offset
BX: Same as AX
ES: ALLOC Address - Segment

DRV_ALLOCVEC returns the address of the allocation vector (ALLOC) for the currently selected drive. If a physical error is encountered when the BDOS Error mode is in one of the return modes (see F_ERRMODE), DRV_ALLOCVEC returns the value 0FFFFH in AX.

Concurrent maintains an allocation vector in memory for each active disk drive. Some programs use the information provided by the allocation vector to determine the amount of free data space on a drive. Note, however, that the allocation information can be inaccurate if the drive has been marked Read-Only.

You can use DRV_SPACE to directly return the number of free 128-byte records on a drive. Concurrent's SHOW utility displays a drive's free space by using the DRV_SPACE call.

DRV_ALLRESET

Restore All Drives To Reset State

Entry Parameters:
Register CL: 0DH (13)

Returned Values:
Register AL: 0 if successful, 0FFH on error
BL: Same as AL

DRV_ALLRESET restores the file system to a reset state where all the disk drives are set to Read-Write (see also DRV_SETRO and DRV_ROVEC), the default disk is set to drive A, and the default DMA address is reset to offset 080H relative to the current DMA segment address.

DRV_ALLRESET can be used, for example, by an application program that requires disk changes during operation. You can also use DRV_RESET for this purpose.

DRV_ALLRESET is conditional under Concurrent, so if another process has a file open on any of the drives to be reset, and the drive is also Read-Only or removable, the DRV_ALLRESET call is denied, and none of the specified drives are reset (see Section 2.17).

Upon return, if the reset operation is successful, DRV_ALLRESET sets register AL to 00H. Otherwise, it sets register AL to 0FFH. If the BDOS is not in one of the return error modes (see F_ERRMODE), the file system displays an error message at the console identifying the process owning the first open file that caused the DRV_ALLRESET to be denied.

DRV_DPB

Return Address Of Disk Parameter Block
For Calling Process's Default Disk

Entry Parameters:
Register CL: 01FH (31)

Returned Values:
Register AX: DPB Address - Offset, 0FFFFH - on Physical Error
BX: Same as AX
ES: DPB Address - Segment

DRV_DPB returns the address of the XIOS-resident **Disk Parameter Block** (DPB) for the currently selected drive. The calling process can use this address to extract the disk parameter values.

If a physical error is encountered when the BDOS Error mode is one of the Return Error modes (see F_ERRMODE), DRV_DPB returns the value 0FFFFH.

Figure 6-4 shows the Disk Parameter Block format. Table 6-11 contains the DPB field defintions.

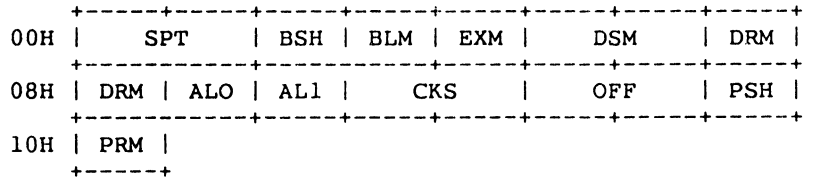


Figure 6-4. DPB - Disk Parameter Block

Table 6-11. DPB Field Definitions

Field	Definition
SPT Sectors Per Track	The number of Sectors Per Track equals the total number of physical sectors per track. Physical sector size is defined by PSH and PRM described below.
BSH Allocation Block Shift Factor	
BLM Allocation Block Mask	The data allocation block size determines the values of the data allocation Block Shift Factor and the allocation Block Mask. The Block Shift factor equals the logarithm base two of the block logical size in 128 byte records, or BSH = LOG2(BLS). The Block Mask equals the number of 128-byte records in an allocation block minus 1, or BLM = (2**BSH)-1. Refer to the System Guide for valid block sizes and BSH and BLM values.
EXM Extent Mask	The data block allocation size and the number of disk allocation blocks determine the value of the Extent Mask. The Extent Mask determines the maximum number of 16k extents that can be contained in a directory entry. It is equal to the maximum number of 16K extents per directory entry minus one. Refer to the System Guide for EXM values.
DSM Disk Storage Maximum	The Disk Storage Maximum defines the total storage capacity of the drive. This is equal to the total number of allocation blocks minus 1 for the drive. DSM must be less than or equal to 7FFFH. If the disk uses 1024 byte blocks (BSH=3, BLM=7), DSM must be less than or equal to 00FFH.
DRM Directory Maximum	The Directory Maximum defines the total number of directory entries for the drive. This is equal to the total number of directory entries, minus 1, that can be kept on this drive. The directory requires 32 bytes of disk per entry. The maximum directory allocation is 16 blocks, where the block size is determined by BSH and BLM.

Table 6-11. (Cont'd)

Field	Definition
AL0	Directory Allocation Vector 0
AL1	Directory Allocation Vector 1 The Directory Allocation Vectors determine the reserved directory allocation blocks.
CKS	Checksum Vector Size The Checksum Vector Size determines the required length of the directory checksum vector and the number of directory entries that the BDOS will checksum. The Checksum Vector Size is equal to the number of directory entries divided by 4, or CKS = (DRM+1)/4. If the media is fixed, CKS might be zero, no storage needs to be reserved, and the BDOS does not calculate directory checksums for the drive. The high-bit of CKS (that is, >= 08000H) is set if the referenced drive is considered to be a nonremovable media drive. Note that this modifies the rules for resetting the drive. For more information, refer to Section 2.15.
OFF	Track Offset The Track Offset is the number of reserved tracks at the beginning of the disk. OFF is equal to the track number on which the directory starts.
PSH	Physical Record Shift Factor The Physical Record Shift Factor ranges from 0 to 5, corresponding to physical record sizes of 128, 256, 512, 1K, 2K, or 4K bytes. It is equal to the logarithm base two of the physical record size divided by 128, or LOG2(sector_size/128).
PRM	Physical Record Mask The Physical Record Mask ranges from 0 to 31, corresponding to physical record sizes of 128, 256, 512, 1K, 2K, or 4K bytes. It is equal to the physical sector size divided by 128 minus 1, or (sector_size/128)-1. For more information on DPB parameters, refer to the System Guide.

DRV_FLUSH

Flush Write-Deferred Buffers

Entry Parameters:
Register CL: 030H (48)
DL: Purge Flag

Returned Values:
Register AL: Error Flag
AH: Permanent Error
BX: Same as AX

DRV_FLUSH forces the write of any write-pending records contained in internal blocking/deblocking buffers. If register DL is set to 0FFH, DRV_FLUSH also purges all active data buffers after performing the writes.

Programs that provide write with read verify support need to purge internal buffers to ensure that verifying reads actually access the disk instead of returning data resident in internal data buffers. Concurrent's PIP utility is an example of such a program.

Upon return, DRV_FLUSH sets register AL to 00H if the flush operation is successful. If a physical error is encountered, DRV_FLUSH performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the error and terminates the calling process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical error codes:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk

DRV_FREE

Free Specified Disk Drives

Entry Parameters:
Register CL: 027H (39)
DX: Drive Vector

DRV_FREE purges the system Lock List of all file and locked record items that belong to the calling process on the specified drives. DRV_FREE passes the drive vector in register DX.

DRV_FREE does not close files associated with purged open file Lock List items. In addition, if a process references a purged file with a BDOS system call requiring an open FCB, DRV_FREE returns a checksum error. A file that has been written to should be closed before making a DRV_FREE call to the file's drive, or data can be lost. Refer to Section 2.17 for more information on DRV_FREE.

Figure 6-3 on page 6-52, shows the format of the Drive Vector.

Note: DRV_FREE treats a floating drive as the physical drive to which it is mapped. For example, if drive N is mapped unto drive A and you call DRV_FREE on A, all files on N are lost in addition to those on A.

DRV_GET

Return The Calling Process's Default Drive

Entry Parameters:
Register CL: 019H (25)

Returned Values:
Register AL: Drive Number
BL: Same as AL

DRV_GET returns the calling process's currently selected default disk number. The disk numbers range from 0 through 15, corresponding to drives A through P.

DRV_GETLABEL

Return Directory Label Data Byte For The Specified Drive

Entry Parameters:
Register CL: 065H (101)
DL: Drive

Returned Values:
Register AL: Directory Label Data Byte
AH: Physical Error
BX: Same as AX

DRV_GETLABEL returns the directory label data byte for the specified drive. The calling process passes the drive number in register DL with 0 for drive A, 1 for drive B, continuing through 15 for drive P in a full 16-drive system.

The directory label data byte has the following format:

bit 7 Require passwords for password protected files
bit 6 Perform access time and date stamping
bit 5 Perform update time and date stamping
bit 4 Perform create time and date stamping
bit 0 Directory label exists on drive

(Bit 0 is the least significant bit)

DRV_GETLABEL returns the directory label data byte in register AL. Register AL equal to 00H indicates that no directory label exists on the specified drive.

If DRV_GETLABEL encounters a physical error when the BDOS Error mode is in one of the return error modes (see F_ERRMODE), it returns with register AL set to 0FFH and register AH set to one of the following:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error

DRV_LOGINVEC

Return Bit Map Of Logged-in Disk Drives

Entry Parameters:
Register CL: 018H (24)

Returned Values:
Register AX: Login Vector
BX: Same as AX

DRV_LOGINVEC returns the Login Vector in register AX. The Login Vector is a 16-bit value with the least significant bit corresponding to drive A, and the high-order bit corresponding to the 16th drive, drive P. A 0 bit indicates the drive is not logged-in, while a 1 bit indicates the drive is logged in.

The Login Vector is identical in format to the Drive Vector show in Figure 6-3, on page 6-52.

DRV_RESET

Reset Specified Disk Drives

Entry Parameters:
Register CL: 025H (37)
DX: Drive Vector

Returned Values:
AL: Return Code
BL: Same as AL

DRV_RESET is used to programmatically restore specified removable media drives to the reset state (a reset drive is not logged in and is in Read-Write status).

Upon entry, register DX contains a 16-bit vector of drives to be reset, where the least significant bit corresponds to drive A, and the high-order bit corresponds to the sixteenth drive, labeled P. Bit values of 1 indicate that the specified drive is to be reset (see Figure 6-3).

DRV_RESET is conditional under Concurrent, so if another process has a file open on any of the drives to be reset, the call is denied, and none of the drives are reset. Refer to Section 2.17 for more information regarding the use of DRV_RESET.

Upon return, if the reset operation is successful, DRV_RESET sets register AL to 00H. Otherwise, it sets register AH to 0FFH. If the BDOS Error mode is not in Return Error mode (see F_ERRMODE), Concurrent displays an error message at the console, identifying the process owning the first open file that caused the DRV_RESET request to be denied.

DRV_ROVEC

Return Bit Map Of Read-Only Disks

Entry Parameters:
Register CL: 01DH (29)

Returned Values:
Register AX: R/O Vector
BX: Same as AX

DRV_ROVEC returns a bit vector indicating which drives have the temporary Read-Only bit set. The Read-Only bit can only be set by a DRV_SETRO call.

Note: When the file system detects a change in the media on a drive, it automatically logs in the drive and sets it to Read-Write.

The format of the R/O Vector is analogous to that of the Login Vector (see Figure 6-3). The least significant bit corresponds to drive A; the most significant bit corresponds to drive P.

DRV_SET

Set Calling Process's Default Disk

Entry Parameters:
Register CL: 0EH (14)
DL: Selected disk

Returned Values:
Register AL: Error Flag
AH: Physical Error
BX: Same as AX

DRV_SET designates the specified disk drive as the default disk for subsequent BDOS file operations. Set the DL register to 0 for drive A, 1 for drive B, continuing through 15 for drive P. DRV_SET also logs in the designated drive if it is currently in the reset state. Logging in a drive activates the drive's directory for file operations.

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

Upon return, register AL equal to 00H indicates the select operation was successful. If a physical error is encountered, DRV_SET performs different actions depending on the BDOS Error mode (see F_ERRMODE).

If the BDOS is in the default Error mode, Concurrent displays a message at the console, identifying the error and terminates the calling process. Otherwise, DRV_SET returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical error codes:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error

DRV_SETLABEL

Create Or Update A Directory Label

Entry Parameters:
- Register CL: 064H (100)
- DX: FCB Address - Offset
- DS: FCB Address - Segment

Returned Values:
- Register AL: Directory Code
- AH: Physical or Extended Error
- BX: Same as AX

DRV_SETLABEL creates a directory label or updates the existing directory label for the specified drive. The calling process passes the address of an FCB containing the name, type, and extent fields to be assigned to the directory label.

The name and type fields of the referenced FCB are not used to locate the directory label in the directory; they are simply copied into the updated or created directory label. Byte 12 of the FCB contains the user's specification of the directory label data byte.

The directory label data byte has the following definition:

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

(Bit 0 is the least significant bit)

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

DRV_SETLABEL also requires that the referenced directory contains SFCBs in order to activate date and time stamping on the drive. If you attempt to activate date and time stamping when no SFCBs exist, DRV_SETLABEL returns an error code and performs no action. Concurrent's INITDIR utility initializes a directory for date and time stamping by placing an SFCB in every fourth entry of the directory.

Upon return, DRV_SETLABEL returns a directory code in register AL with the value 00H if the directory label create or update was successful, or 0FFH if no space existed in the referenced directory to create a directory label. It also returns 0FFH if date and time stamping was requested and the referenced directory did not contain SFCBs. Register AH is set to 00H in all of these cases.

If a physical or extended error is encountered, DRV_SETLABEL performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the error and terminates the calling process. Otherwise, DRV_SETLABEL returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended error codes:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
04H - Invalid Drive : drive select error
07H - Password Error

DRV_SETRO

Set Default Disk To Read-Only

Entry Parameters:
Register CL: 01CH (28)

Returned Values:
Register AL: Return Code
BL: Same as AL

DRV_SETRO provides temporary write protection for the currently selected disk by marking the drive as Read-Only. No process can write to a disk that is in the Read-Only state. You must perform a successful DRV_RESET operation to restore a Read-Only drive to the Read-Write state (see DRV_ALLRESET and DRV_RESET).

DRV_SETRO is conditional under Concurrent, so if another process has an open file on the drive, the operation is denied, and DRV_SETRO returns the value 0FFH to the calling process. Otherwise, it returns a 00H.

If the BDOS is not in Return Error mode (see F_ERRMODE), Concurrent displays an error message at the console, identifying the process owning the first open file that caused the DRV_SETRO request to be denied.

Note that a drive in the Read-Only state cannot be reset by a process if another process has an open file on the drive.

DRV_SPACE

Return Free Disk Space On Specified Drive

Entry Parameters:
Register CL: 02EH (46)
DL: Drive

Returned Values:
Register AL: Error Flag
AH: Physical Error
BX: Same as AX, First 3 bytes of DMA Buffer filled in

DRV_SPACE determines the number of free sectors (128-byte records) on the specified drive. The calling process passes the drive number in register DL, with 0 for drive A, 1 for B, continuing through 15 for drive P.

DRV_SPACE returns a binary number in the first 3 bytes of the current DMA buffer. Figure 6-5 shows the format of the returned number.

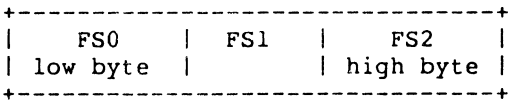


Figure 6-5. Disk Free Space Field Format

Note that the returned free space value might be inaccurate if the drive has been marked Read-Only.

Upon return, DRV_SPACE sets register AL to 00H, indicating the operation was successful. However, if the BDOS is in one of the return Error modes (see F_ERRMODE), and a physical error occurs, it sets register AL to 0FFH, and register AH to one of the following values:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error

F_ATTRIB

Set The Attributes Of A Disk File

Entry Parameters:
Register CL: 01EH (30)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Directory Code
BL: Same as AL

F_ATTRIB can modify a file's attributes and set its last record byte count. Other BDOS system calls can interrogate these file parameters, but only F_ATTRIB can change them.

F_ATTRIB can set or reset the file attributes: F1' through F4', Read-Only (T1'), System (T2'), and Archive (T3').

The specified FCB contains a filename with the appropriate attributes set or reset. The calling process must ensure that it does not specify an ambiguous filename. Also, if the specified file is password protected, the correct password must be placed in the first eight bytes of the current DMA buffer or have been previously established as the default password (see F_PASSWD).

Interface attribute F5' specifies whether an extended file lock is to be maintained after the F_ATTRIB call. Interface attribute F6' specifies if the specified file's byte count is to be set. The interface attribute definitions are listed below:

F5'= 0	Do not maintain an extended file lock (default)
F5'= 1	Maintain an extended file lock
F6'= 0	Do not set byte count (default)
F6'= 1	Set byte count

If F5' is set and the referenced FCB specifies a file with an extended file lock, the calling process maintains the lock on the file. Otherwise, the file becomes available to other processes. Section 2.11 describes extended file locking in detail.

If interface attribute F6' is set, the calling process must set the CR field of the referenced FCB to the new byte count value. A process can access a file's byte count value with the BDOS F_OPEN, F_SFIRST, and F_SNEXT system calls. File byte counts are described in Section 2.15.

F_ATTRIB searches the FCB specified directory for an entry belonging to the current user number that matches the FCB specified name and type fields. F_ATTRIB then updates the directory to contain the selected indicators, and if interface attribute F6' is set, the specified byte count value. Note that the last record byte count is maintained in the byte 13 of a file's directory FCBs.

File attributes T1', T2', and T3' are defined by Concurrent as described in Section 2.4.2. Attributes F1' through F4' of command files are defined as Compatibility Attributes, as described in Section 2.12. However, for all other files, attributes F1' through F4' can be redefined. Attributes F5' through F8' are reserved as Interface Attributes and cannot be used as file attributes. Interface attributes are described in Section 2.4.3.

An F_ATTRIB call is not performed if the referenced FCB specifies a file currently open for another process. It is performed, however, if the referenced file is open by the calling process in Locked mode. However, the file's lock entry is purged when this is done and the file system prevents continued read and write operations on the file. F_ATTRIB does not set the attributes of a file currently open in Read-Only or Unlocked mode for any process.

Making an F_ATTRIB call for an open file can adversely affect the performance of the calling process. For this reason, you should close an open file before you call F_ATTRIB.

Upon return, F_ATTRIB returns a directory code in register AL with the value 00H if the call is successful, or 0FFH if the file specified by the referenced FCB is not found. Register AH is set to 00H in both cases.

If a physical or extended error is encountered, F_ATTRIB performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the error and terminates the process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended error codes:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
04H - Invalid Drive : drive select error
05H - File open by another process
07H - Password Error
09H - Illegal ? in FCB

F_CLOSE

Close A Disk File

Entry Parameters:
- Register CL: 010H (16)
- DX: FCB Address - Offset
- DS: FCB Address - Segment

Returned Values:
- Register AL: Directory Code
- AH: Physical or Extended Error
- BX: Same as AX

F_CLOSE performs the inverse operation of F_OPEN. The referenced FCB must have been previously activated by a successful F_OPEN or F_MAKE call. Interface attributes F5' and F6' specify how the file is to be closed, as shown below:

- F5' = 0, F6' = 0 Default Close
- F5' = 0, F6' = 1 Extend File Lock
- F5' = 1, F6' = 0 Partial Close
- F5' = 1, F6' = 1 Partial Close

F_CLOSE performs the following steps regardless of the interface attribute specification.

1. First, it verifies the referenced FCB has a valid checksum. If the checksum is invalid, F_CLOSE performs no action and returns an error code.
2. If the checksum is valid and the referenced FCB contains new information because of write operations to the FCB, F_CLOSE permanently records the new information in the directory. If the FCB does not contain new information, the directory update step is bypassed. However, F_CLOSE always attempts to locate the FCB's corresponding entry in the directory and returns an error code if the directory entry cannot be found.

If F_CLOSE successfully performs the above steps, it performs different actions, depending on how the interface attributes are set.

In default close operations, F_CLOSE decrements the file's open count, which is maintained in the file's system Lock List entry. If the open count decrements to zero, it indicates that the number of default close operations for the file matches the number of open operations.

If the open count decrements to zero, F_CLOSE permanently closes the file by performing the following steps.

1. First, it removes the file's item from the system Lock List. If the FCB is opened in Unlocked mode, it also purges all record locks belonging to the file from the system Lock List.
2. In addition, F_CLOSE invalidates the FCB's checksum to ensure the referenced FCB is not subsequently used with BDOS system calls that require an open FCB (for example, F_WRITE).
3. If the open count does not decrement to zero, F_CLOSE simply returns to the calling process and the file remains open.

For partial close operations, F_CLOSE does not decrement the file's open count and returns to the calling process. The file always remains open following a partial close request.

Closing a file with an extended file lock modifies the way F_CLOSE performs a permanent close. F_CLOSE only honors an extended lock request on a permanent close of a file opened in Locked Mode. If these conditions are satisfied, F_CLOSE invalidates the FCB's checksum but maintains the lock item. Thus, although the file is permanently closed, other processes cannot access the file. Section 2.11 describes extended file locking in detail.

Upon return, F_CLOSE returns a directory code in register AL with the value 00H if the close operation is successful, or 0FFH if the file is not found. Register AH is set to 0 in both of these cases.

If a physical or extended error is encountered, F_CLOSE performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message identifying the error at the console and terminates the calling process. Otherwise F_CLOSE returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended error codes:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
04H - Invalid Drive : drive select error
06H - Close Checksum Error

F_DELETE

Delete A Disk File

Entry Parameters:
- Register CL: 013H (19)
- DX: FCB Address - Offset
- DS: FCB Address - Segment

Returned Values:
- Register AL: Directory Code
- AH: Physical or Extended Error
- BX: Same as AX

F_DELETE removes files and/or XFCBs that match the FCB addressed in register DX. The filename and filetype fields can contain wildcard file specifications (question marks in bytes 1 through 11), but byte 0 cannot be a wildcard as it can be in the F_SFIRST and F_SNEXT calls.

Interface attribute F5' specifies the type of delete operation to be performed, as shown below:

- F5' = 0 Standard Delete (Default mode)
- F5' = 1 Delete only XFCB's and maintain an extended file lock.

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

For standard delete operations, F_DELETE removes all directory entries belonging to files that match the referenced FCB. All disk directory and data space owned by the deleted files is returned to free space and becomes available to other files. Directory XFCBs that were owned by the deleted files are also removed from the directory. If interface attribute F5' of the FCB is set to 1, F_DELETE deletes only the directory XFCBs matching the referenced FCB.

Note: If any of the files matching the input FCB specification fail the password check, are Read-Only, or are currently open by another process, then F_DELETE deletes no files or XFCBs. This applies to both types of delete operations.

Interface attribute F5' also specifies whether an extended file lock is to be maintained after the F_DELETE call. If F5' is set and the referenced FCB specifies a file with an extended lock, the calling process maintains the lock on the file. Section 2.11 describes extended file locking in detail.

A process can delete a file that it currently has open if the file is opened in locked mode. However, the BDOS returns a checksum error if the process makes a subsequent reference to the file with a BDOS system call requiring an open FCB. A process cannot delete files open in Read-Only or Unlocked mode.

Deleting an open file can adversely affect the performance of the calling process. For this reason, you should close an open file before you delete it.

Upon return, F_DELETE returns a directory code in register AL with the value 00H if the delete is successful, or 0FFH if no file matching the referenced FCB is found. Register AH is set to 0 in both of these cases.

If a physical or extended error is encountered, F_DELETE performs different actions, depending on the BDOS Error mode (see F_ERRMODE).

If the BDOS is the default Error mode, Concurrent displays a message identifying the error at the console and terminates the calling process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended error codes:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
03H - Read/Only File
04H - Invalid Drive : drive select error
05H - File opened by another process or
open in Read-Only or Unlocked mode
07H - Password Error

F_DMAGET

Return Address Of Direct Memory Access Buffer

Entry Parameters:
Register CL: 034H (52)

Returned Values:
Register AX: DMA Offset
BX: Same as AX
ES: DMA Segment

F_DMAGET returns the current DMA Base Segment address in ES, with the current DMA Offset in AX.

DMA is an acronym for **Direct Memory Address**, which is often used with disk controllers that directly access the memory of the computer to transfer data to and from the disk subsystem.

Under Concurrent, the current DMA is usually defined as the buffer in memory where a record resides before a disk write and after a disk read operation. If the BDOS Multisector Count is equal to one (see F_MULTISEC), the size of the buffer is 128 bytes. However, if the BDOS Multisector Count is greater than one, the size of the buffer must equal N * 128, where N equals the Multisector Count.

Some BDOS system calls also use the current DMA to pass parameters and to return values. For example, BDOS system calls that check and assign file passwords require that the password be placed in the current DMA Buffer. As another example, DRV_SPACE returns its results in the first 3 bytes of the current DMA. When the current DMA is used in this context, the size of the buffer in memory is determined by the specific requirements of the system call.

F_DMAOFF

Set The Direct Memory Address Offset

Entry Parameters:
Register CL: 01AH (26)
DX: DMA Address - Offset

F_DMAOFF can change the default value of the DMA offset to another memory address. When P_CLI initiates a transient program, it sets the DMA offset to 080H and the DMA Segment or Base to its initial Data Segment. DRV_ALLRESET also sets the DMA offset to 080H. The DMA address remains at its current value until it is changed by an F_DMASEG, F_DMAOFF, or DRV_ALLRESET call.

F_DMASEG

Set Direct Memory Access Segment Address

Entry Parameters:
Register CL: 033H (51)
DX: DMA Segment Address

F_DMASEG sets the segment value of the current DMA buffer address. The word parameter in DX is a paragraph address and is used with the DMA offset value to specify the 20-bit address of the DMA buffer.

Note that upon initial program loading, the default DMA base is set to the address of the user's data segment (the initial value of DS) and the DMA offset is set to 0080H, which provides access to the default buffer in the Base Page.

F_ERRMODE

Set BDOS Error Mode For Error Returns

Entry Parameters:
Register CL: 02DH (45)
DL: BDOS Error mode

F_ERRMODE sets the BDOS **Error mode**, which is a system parameter maintained for each running process that determines how the file system handles physical and extended errors. Physical and extended errors are described in Section 2.18.

The BDOS Error mode has three states: the default mode, Return and Error mode, and Return and Display mode.

The BDOS performs different actions in each mode when a physical or extended error occurs:

* In the default Error mode, the BDOS displays a system message at the console identifying the error and terminates the calling process.
* In Return Error mode, the BDOS sets register AL to 0FFH, places an error code identifying the physical or extended error in register AH, and returns to the calling process.
* In Return and Display mode, the BDOS displays the system message before returning to the calling process, and sets registers AH and AL as in the Return Error mode.

F_ERRMODE sets the BDOS Error mode as specified in register DL. If register DL is set to 0FFH, the mode is set to Return Error mode. If register DL is set to 0FEH, the mode is set to Return and Display mode. If register DL is set to any other value, the mode is set to the default mode.

F_LOCK

Lock Records in a Disk File

Entry Parameters:
Register CL: 02AH (42)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Error Code
AH: Physical Error
BX: Same as AX

F_LOCK allows a process to establish temporary ownership of particular records within a file, and is only supported for files open in Unlocked mode. If F_LOCK is called for a file open in Locked or Read-Only mode, no locking action is performed but a successful result is returned. This provides compatibility between Concurrent and CP/M-86.

The calling process passes the address of an FCB in which the random record field is filled with the Random Record Number of the first record to be locked. The number of records to be locked is determined by the BDOS Multisector Count (see F_MULTISEC). The current DMA must also contain the 2-byte File ID returned by F_OPEN or F_MAKE when the referenced FCB was opened. Note that the File ID is only returned by F_OPEN and F_MAKE when the Open mode is Unlocked.

Interface attribute F5' specifies the type of lock to perform. Interface attribute F6' specifies whether records have to exist in order to be locked. The F_LOCK interface attribute definitions are listed below:

F5'= 0 Exclusive lock (default)
F5'= 1 Shared lock
F6'= 0 Lock existing records only (default)
F6'= 1 Lock logical records.

These options are described in detail in Section 2.14.

F_LOCK verifies that a locking conflict with another process does not exist for each of the records to be locked. In addition, if F_LOCK is called with attribute F6' reset, it also verifies that each record number to be locked exists within the specified file. Both tests are made before any records are locked.

Most F_LOCK requests require a new entry in the BDOS system Lock List. If there is insufficient space in the system Lock List to satisfy the lock request, or if the process record lock limit is exceeded, then F_LOCK does not lock any records and returns an error code to the calling process.

Upon return, F_LOCK sets register AL to 00H if the lock operation is successful. Otherwise, register AL contains one of the following error codes:

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

F_LOCK returns error code 01H when it accesses a data block that has not been previously written.

F_LOCK returns error code 03H when it cannot close the current extent prior to moving to a new extent.

F_LOCK returns error code 04H when it accesses an extent that has not been created.

F_LOCK returns error code 06H when byte 35 (R2) of the referenced FCB is greater than 3.

F_LOCK returns error code 08H if the specified record is locked by another process with an incompatible lock type.

F_LOCK returns error code 0AH if the referenced FCB failed the FCB checksum test.

F_LOCK returns error code 0BH if the BDOS cannot locate the referenced FCB's directory entry when attempting to verify that the FCB contains current information.

F_LOCK returns error code 0CH if performing the lock request would require that the process consume more than the maximum allowed number of system Lock List entries.

F_LOCK returns error code 0DH when an invalid File ID is placed at the beginning of the current DMA.

F_LOCK returns error code 0EH when the system Lock List is full and performing the lock request would require at least one new entry.

F_LOCK returns error code 0FFH if a physical error is encountered, and the BDOS Error mode is either Return Error mode or Return and Display Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the physical error and terminates the calling process. When F_LOCK returns a physical error to the calling process, it is identified by register AH as shown below:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error

F_MAKE

Create A Disk File

Entry Parameters:
Register CL: 016H (22)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Directory Code
AH: Physical or Extended Error
BX: Same as AX

F_MAKE creates a new directory entry for a file under the current user number. It also creates an XFCB for the file if the referenced drive has a directory label that enables password protection on the drive, and the calling process assigns a password to the file.

The calling process passes the address of the FCB with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the filename and filetype, and byte 12 set to the extent number. Byte 12, the EX field, is usually set to 00H. Byte 32 of the FCB, the CR field, must be initialized to 00H, before or after the F_MAKE call, if the intent is to write sequentially from the beginning of the file.

Interface attribute F5' specifies the mode in which the file is to be opened. Interface attribute F6' specifies whether a password is to be assigned to the created file. The interface attributes are summarized below:

F5' = 0 Open in Locked mode (default)
F5' = 1 Open in Unlocked mode
F6' = 0 Do not assign password (default)
F6' = 1 Assign password to created file

When attribute F6' is set to 1, the calling process must place the password in the first 8 bytes of the current DMA buffer and set byte 9 of the DMA buffer to the password mode. Note that F_MAKE only interrogates attribute F6' if the referenced drive's directory label has enabled password support. The XFCB Password mode is summarized below:

Bit 7 Read mode
Bit 6 Write mode
Bit 5 Delete mode

F_MAKE returns with an error code if the referenced FCB names a file that currently exists in the directory under the current user number. If there is any possibility of duplication, an F_DELETE call should precede the F_MAKE call.

If the make file operation is successful, F_MAKE activates the referenced FCB for record operations (opens the FCB) and initializes both the directory entry and the referenced FCB to an empty file.

F_MAKE also computes a checksum and assigns it to the FCB. BDOS system calls that require an open FCB (for example, F_WRITE) verify that the FCB checksum is valid before performing their operation.

If the file is opened in Unlocked mode, F_MAKE also sets bytes R0 and R1 in the FCB to a two-byte value called the **File ID**. The File ID is a required parameter for the BDOS Lock Record and Unlock Record system calls. Note that F_MAKE initializes all file attributes to 0.

The BDOS also creates an open file item in the system Lock List to record a successful F_MAKE operation. While this item exists, no other process can delete, rename, truncate, or set the file attributes of this file.

A creation and/or update stamp is made for the created file if the referenced drive contains a directory label that enables creation and/or update time and date stamping and the FCB extent number is equal to 0.

F_MAKE also creates an XFCB for the created file if the referenced drive contains a directory label that enables password protection, interface attribute F6' of the FCB is 1, and the FCB is an extent zero FCB. In addition, F_MAKE also assigns the password and password mode placed in the first nine bytes of the DMA to the XFCB.

Upon return, F_MAKE returns a directory code in register AL with the value 00H if the make operation is successful, or 0FFH if no directory space is available. Register AH is set to 00H in both cases.

If a physical or extended error is encountered, F_MAKE performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the error and terminates the calling process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended error codes:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
04H - Invalid Drive : drive select error
08H - File Already Exists
09H - Illegal ? in FCB
0AH - Open File Limit Exceeded
0BH - No Room in System Lock List

F_MULTISEC

Set BDOS Multisector Count

Entry Parameters:
Register CL: 02CH (44)
DL: Number of Sectors

Returned Values:
Register AL: Return Code
BL: Same as AL

F_MULTISEC provides logical record blocking under Concurrent. It enables a process to read and write from 1 to 128 physical records of 128 bytes at a time during subsequent BDOS read and write system calls. It also specifies the number of 128-byte records to be locked or unlocked by F_LOCK and F_UNLOCK.

F_MULTISEC sets the **Multisector Count** value for the calling process to the value passed in register DL. Once set, the specified Multisector Count remains in effect until the calling process makes another F_MULTISEC call and changes the value. Note that P_CLI sets the Multisector Count to one when it initiates a transient program.

The Multisector Count affects BDOS error reporting for the BDOS read and write system calls. With the exception of physical errors, if an error occurs during these system calls and the Multisector Count is greater than one, Concurrent returns the number of records successfully processed in register AH.

Upon return, F_MULTISEC sets register AL to 00H if the specified value is in the range of 1 to 128. Otherwise, it sets register AL to 0FFH.

F_OPEN

Open A Disk File

Entry Parameters:
- Register CL: 0FH (15)
- DX: FCB Address - Offset
- DS: FCB Address - Segment

Returned Values:
- Register AL: Directory Code
- AH: Physical or Extended Error
- BX: Same as AX

F_OPEN activates the FCB for a file that exists in the disk directory under the currently active user number or user zero. The calling process passes the address of the FCB, with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the filename and filetype, and byte 12 specifying the extent. Byte 12 is usually set to zero.

Interface attributes F5' and F6' of the FCB specify the mode in which the file is to be opened, as shown below:

F5' = 0, F6' = 0	Open in locked mode (Default mode)
F5' = 1, F6' = 0	Open in Unlocked mode
F5' = 0 or 1, F6' = 1	Open in Read-Only mode

If the file is password protected in Read mode, the correct password must be placed in the first eight bytes of the current DMA or have been previously established as the default password (see F_PASSWD). If the current record field of the FCB, CR, is set to 0FFH, F_OPEN returns the byte count of the last record of the file in the CR field. The last record byte count for a file can be set using F_ATTRIB.

Note: The calling process must set the CR field of the FCB to 00H if the file is to be accessed sequentially from the first record.

F_OPEN performs the following steps for files opened in locked or Read-Only mode. If the current user is nonzero and the file to be opened does not exist under the current user number, F_OPEN searches user 0 for the file. If the file exists under user 0 and has the system attribute (T2') set, the file is opened under user 0. The Open mode is automatically set to Read-Only when this is done.

F_OPEN also performs the following action for files opened in locked mode when the current user number is 0. If the file exists in the directory under user 0, and has both the system attribute (T2') set and the Read-Only attribute (T1') set, the Open mode is automatically set to Read-Only. Note that Read-Only mode implies the file can be concurrently accessed by other processes if they also open the file in Read-Only mode.

If the open operation is successful, F_OPEN activates the user's FCB for record operations as follows: F_OPEN copies the relevant directory information from the matching directory FCB into bytes D0 through D15 of the FCB. It also computes a checksum and assigns it to the FCB. All BDOS system calls that require an open FCB (for example, F_READ) verify that the FCB checksum is valid before performing their operation.

If the file is opened in Unlocked mode, F_OPEN sets bytes R0 and R1 of the FCB to a two-byte value called the **File ID**. The File ID is a required parameter for the F_LOCK and F_UNLOCK calls. If the Open mode is forced to Read-Only, F_OPEN sets interface attribute F8' to 1 in the user's FCB. In addition, the system call sets attribute F7' to 1 if the referenced file is password protected in Write mode and the correct password was not passed in the DMA or did not match the default password. The BDOS does not support write operations for an activated FCB if interface attribute F7' or F8' is set to 1.

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

When the open operation is successful, F_OPEN also makes an access time and date stamp for the opened file when the following conditions are satisfied: the referenced drive has a directory label that requests access date and time stamping, the FCB extent field is equal to zero, and the referenced drive is Read-Write.

Upon return, F_OPEN returns a directory code in register AL with the value 00H if the open is successful, or 0FFH if the file is not found. Register AH is set to 0 in both of these cases. If a physical or extended error is encountered, F_OPEN performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message identifying the error at the console and terminates the process. Otherwise, F_OPEN returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended error codes:

- 01H - Disk I/O Error : permanent error
- 04H - Invalid Drive : drive select error
- 05H - File is open by another process or by the current process in an incompatible mode
- 07H - Password Error
- 09H - Illegal ? in FCB
- 0AH - Open File Limit Exceeded
- 0BH - No Room in System Lock List

F_PARSE

Parse An ASCIi String And Initialize An FCB

Entry Parameters:
- Register CL: 098H (152)
- DX: PFCB Address - Offset
- DS: PFCB Address - Segment

Returned Values:
- Register AX: 0FFFFH if error
  0 if end of filename string
  0 if end of line address of next item to parse
- BX: Same as AX
- CX: Error Code

F_PARSE parses an ASCII file specification and prepares a File Control Block. The calling process passes the address of a data structure called the **Parse Filename Control Block**, (PFCB) in registers DX and DS.

Figure 6-6 shows the format of the Parse Filename Control Block. Table 6-12 lists the fields in the PFCB.

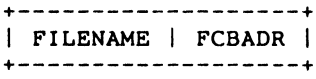


Figure 6-6. PFCB - Parse Filename Control Block

Table 6-12. PFCB Field Defintions

Field	Description
FILENAME	Offset of an ASCII file specification to parse. The offset is relative to the same Data Segment as the PFCB.
FCBADR	Offset of a File Control Block to initialize. The offset is relative to the same Data Segment as the PFCB.

F_PARSE assumes the file specification to be in the following form:

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

where those items enclosed in curly brackets are optional.

F_PARSE parses the first file specification it finds in the input string. First of all, it eliminates leading blanks and tabs. F_PARSE then assumes the file specification ends on the first delimiter it encounters that is out of context with the specific field it is parsing. For instance, if it finds a colon (:), and it is not the second character of the file specification, the colon delimits the whole file specification.

F_PARSE recognizes the following characters as delimiters:

space
tab
carriage return
null
; (semicolon) - except before password field
= (equal)
< (less than)
> (greater than)
. (period) - except after filename and before filetype
: (colon) - except before filename and after drive
, (comma)
| (vertical bar)
[(left square bracket)
] (right square bracket)

If F_PARSE encounters a nongraphic character in the range 1 through 31 not listed above, it treats the character as an error.

F_PARSE initializes the specified FCB as shown in Table 6-13.

Table 6-13. FCB Initialization

Byte	Definition
byte 0	The drive field is set to the specified drive. If the drive is not specified, the default value is used. 0 = default, 1 = A, 2 = B, etc.
byte 1-8	The name is set to the specified filename. All letters are converted to upper-case. If the name is not eight characters long, the remaining bytes in the filename field are padded with blanks. If the filename has an asterisk (*), all remaining bytes in the filename field are filled in with question marks (?). F_PARSE returns an error if the filename is more than eight bytes long.
byte 9-11	The type is set to the specified filetype. If no type is specified, the type field is initialized to blanks. All letters are converted to upper-case. If the type is not three characters long, the remaining bytes in the filetype field are padded with blanks. If an asterisk is encountered, all remaining bytes are filled in with question marks. F_PARSE returns an error if the type field is more than 3 bytes long.
byte 12-15	Filled in with zeros.
byte 16-23	The password field is set to the specified password. If no password is specified, this field is initialized to blanks. If the password is not eight characters long, remaining bytes are padded with blanks. All letters are converted to upper-case. F_PARSE returns an error if the password field is more than eight bytes long.
byte 24-31	Reserved for system use.

If an error occurs, F_PARSE returns 0FFFFH in register AX indicating the error.

On a successful parse, F_PARSE checks the next item in the FILENAME string. It scans for the first character that follows trailing blanks and tabs. If the character is a line feed (0AH), a carriage return (0DH), or a null character (00H), it returns a 0 indicating the end of the FILENAME string. If the next character is a delimiter, it returns the address of the delimiter. If the next character is not a delimiter, it returns the address of the first trailing blank or tab.

If F_PARSE is to be used to parse a subsequent filename in the FILENAME string, the returned address should be advanced over the delimiter before placing it in the PFCB.

Table 6-5 contains the list of error codes returned in CX.

F_PASSWD

Establish A Default Password For File Access

Entry Parameters:
Register CL: 06AH (106)
DX: Password Address - Offset
DS: Password Address - Segment

F_PASSWD allows a process to specify a password value before a file protected by the password is accessed. When the file system accesses a password-protected file, it checks the current DMA, and the default password for the correct value. If either value matches the file's password, full access to the file is allowed.

Concurrent maintains a default password for each process running on the system. A new process inherits its initial default password from its parent, the process creating the new process.

Note: Changing the default password does not affect other processes currently running on the system.

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

F_RANDREC

Return The Random Record Number Of The
Next Record To Access In A Disk File

Entry Parameters:
Register CL: 024H (36)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Random Record Field of FCB Set

F_RANDREC returns the Random Record Number of the next record to be accessed from a file that has been read or written sequentially to a particular point. F_RANDREC returns this value in the Random Record field, bytes R0, R1, and R2, of the addressed FCB. F_RANDREC can be useful in two ways.

First, it is often necessary to initially read and scan a sequential file to extract the positions of various key fields. As each key is encountered, you call F_RANDREC to compute the random record position for the data corresponding to this key. If the data unit size is 128 bytes, the resulting record number minus one is placed into a table with the key for later retrieval.

After scanning the entire file and tabularizing the keys and their record numbers, you can move directly to a particular record by performing a random read using the corresponding Random Record Number that was saved earlier. The scheme is easily generalized when variable record lengths are involved, because the program need only store the buffer-relative byte position along with the key and record number in order to find the exact starting position of the keyed data at a later time.

F_RANDREC can also be used when switching from a sequential read or write to a random read or write. Access records sequentially to a particular point in the file, call F_RANDREC to set the record number, and then subsequent random read and write operations can continue from the next record in the file.

F_READ

Read Records Sequentially From A Disk File

Entry Parameters:
Register CL: 014H (20)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Error Code
AH: Physical Error
BX: Same as AX

F_READ reads the next 1 to 128 128-byte records from a file into memory, beginning at the current DMA address. The BDOS Multisector Count (see F_MULTISEC) determines the number of records to be read. The default is one record. The addressed FCB must have been previously activated by an F_OPEN or F_MAKE call.

F_READ reads each record from the current record (CR) field in the FCB, relative to the current extent, then automatically increments the CR field to the next record position. If the CR field overflows, then F_READ automatically opens the next logical extent and resets the CR field to zero for the next read operation. The calling process must set the CR field to 00H following the open call if the intent is to read sequentially from the beginning of the file.

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

01H - Reading unwritten data (end-of-file)
08H - Record locked by another process
09H - Invalid FCB
0AH - FCB Checksum Error
0BH - Unlocked file verification error
0FFH - Physical error; refer to register AH

F_READ returns error code 01H if no data exists at the next record position of the file. The no data situation is usually encountered at the end of a file. However, it can also occur if you try to read a data block that has not been previously written or an extent that has not been created. These situations are usually restricted to files created or appended with the BDOS random write calls (F_WRITERAND and F_WRITEZF).

F_READ returns error code 08H if the calling process attempts to read a record locked by another process with an exclusive lock. This error code is only returned for files opened in Unlocked mode.

F_READ returns error code 09H if the FCB is invalidated by a previous F_CLOSE call that returned an error.

F_READ returns error code 0AH if the referenced FCB failed the FCB checksum test.

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

F_READ returns error code 0FFH if a physical error is encountered and the BDOS Error mode is in one of the return modes (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the physical error and terminates the calling process. When F_READ returns a physical error to the calling process, it is identified by register AH as shown below:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error

On all error returns, except for physical error returns (AL = 255), F_READ sets register AH to the number of records successfully read before the error was encountered. This value can range from 0 to 127 depending on the current BDOS Multisector Count. It is always set to zero when the Multisector Count is equal to one.

F_READRAND

Read Random Records From A Disk File

Entry Parameters:
Register CL: 021H (33)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Error Code
AH: Physical Error
BX: Same as AX

F_READRAND is similar to F_READ except that the read operation takes place at a particular Random Record Number, selected by the 24-bit value constructed from the three-byte, R0, R1, R2, field beginning at position 33 of the FCB. Note that the sequence of 24 bits is stored with the least significant byte first, R0, the middle byte next, R1, and the high byte last, R2. The Random Record Number can range from 0 to 262,143. This corresponds to a maximum value of 3 in byte R2.

To read a file with F_READRAND, the calling process must first open the base extent, extent 0. This ensures that the FCB is properly initialized for subsequent random access operations. The base extent might or might not contain any allocated data.

F_READRAND reads the record specified by the random record field into the current DMA address. F_READRAND automatically sets the FCB extent and current record number values, EX and CR, but unlike F_READ, it does not advance the current record number. Thus, a subsequent F_READRAND call rereads the same record. After a random read operation, a file can be accessed sequentially, starting from the current randomly accessed position. However, the last randomly accessed record is reread or rewritten when switching from random to sequential mode.

If the BDOS Multisector count is greater than one (see F_MULTISEC), F_READRAND reads multiple consecutive records into memory beginning at the current DMA.

F_READRAND automatically increments the R0, R1, R2 field of the FCB to read each record. However, it restores the FCB's Random Record Number to the first record's value upon return to the calling process.

Upon return, F_READRAND sets register AL to 00H if the read operation is successful. Otherwise, register AL contains one of the following error codes:

01H - Reading unwritten data
03H - Cannot close current extent
04H - Seek to unwritten extent
06H - Random record number out of range
08H - Record locked by another process
0AH - FCB Checksum Error
0BH - Unlocked file verification error
0FFH - Physical error refer to register AH

F_READRAND returns error code 01H when it accesses a data block not previously written. This may indicate an end-of-file condition.

F_READRAND returns error code 03H when it cannot close the current extent prior to moving to a new extent.

F_READRAND returns error code 04H when a read random operation accesses an extent that has not been created.

F_READRAND returns error code 06H when byte 35 (R2) of the referenced FCB is greater than 3.

F_READRAND returns error code 08H if the calling process attempts to read a record locked by another process with an exclusive lock. This error code is only returned for files opened in Unlocked mode.

F_READRAND returns error code 0AH if the referenced FCB failed the FCB checksum test.

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

F_READRAND returns error code 0FFH if a physical error is encountered and the BDOS Error mode is in one of the return modes (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the physical error and terminates the calling process.

When a physical error is returned to the calling process, it is identified by the four low-order bits of register AH as shown below:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error

On all error returns except for physical error returns, AL = 255, F_READRAND sets register AH to the number of records successfully read before the error was encountered. This value can range from 0 to 127 depending on the current BDOS Multisector Count. It is always set to zero when the Multisector Count is equal to one.

F_RENAME

Rename A Disk File

Entry Parameters:
Register CL: 017H (23)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Directory Code
AH: Physical or Extended Error
BX: Same as AX

F_RENAME uses the referenced FCB to change all directory entries of the file specified by the drive and filename in bytes 0 to 11 of the FCB to the filename specified in bytes 17 through 27.

If the file specified by the first filename is password-protected, the correct password must be placed in the first eight bytes of the current DMA buffer, or have been previously established as the default password (see F_PASSWD).

The calling process must also ensure that the filenames specified in the FCB are valid and unambiguous, and that the new filename does not already exist on the drive. F_RENAME uses the drive code at byte 0 of the FCB to select the drive. The drive code at byte 16 of the FCB is ignored.

Interface attribute F5' specifies whether an extended file lock is to be maintained after the F_ATTRIB call as shown below:

F5'= 0 Do not maintain an extended file lock (default)
F5'= 1 Maintain an extended file lock

If F5' is set and the referenced FCB specifies a file with an extended file lock, the calling process maintains the lock on the file. Otherwise, the file becomes available to other processes on the system. Section 2.11 describes extended file locking in detail.

A process can rename a file that it has open if the file is open in locked mode. However, the BDOS returns a checksum error if the process subsequently references the file with a system call requiring an open FCB. A file open in Read-Only or Unlocked mode cannot be renamed by any process.

Renaming an open file can adversely affect the performance of the calling process. For this reason, you should close an open file before you rename it.

Upon return, F_RENAME returns a directory code in register AL with the value 00H if the rename is successful, or 0FFH if the file named by the first filename in the FCB is not found. Register AH is set to 00H in both of these cases.

If a physical or extended error is encountered, F_RENAME performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the error, and terminates the process. Otherwise, it returns to the calling process with register AL set to 0FFH and with register AH set to one of the following physical or extended error codes:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
03H - Read/Only File
04H - Invalid Drive : drive select error
05H - File open by another process
07H - Password Error
08H - File Already Exists
09H - Illegal ? in FCB

F_SETDATE

Set File Time and Date Stamps

Entry Parameters:
Register CL: 74H (116)
DX: FCB address - Offset
DS: FCB address - Segment

Returned Values:
Register AL: Directory code
AH: Physical error
BX: Same as AX

F_SETDATE sets the time and date stamp fields for the specified file to the time and date stamp values specified in the first eight bytes of the DMA buffer. The specified file must currently be open in Locked mode by the calling process.

The first 4-byte field in the DMA buffer contains the access or create stamp field for CP/M™ media files. This field is copied into the file's access or create stamp field if the directory label has activated access and/or creation time and date stamping on the file's drive.

Note that only the update stamp field can be set for DOS media files. DOS media files are not stamped for access or create times.

The second 4-byte field of the DMA buffer contains the update stamp field. This field is copied into the update stamp field for CP/M files only when the directory label has activated update time and date stamping on the file's drive. The DMA update stamp field is always copied into the update stamp field of DOS media files.

Upon return from a successful operation, F_SETDATE sets register AL to 00H. If the referenced FCB does not specify a file opened by the calling process in Locked mode, register AL will be set to 0AH. Register AH is set to 00H in both cases.

If a physical or extended error is encountered, F_SETDATE performs different actions, depending upon the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays an error message at the console and terminates the calling process. Otherwise, F_SETDATE returns to the calling process with register AL set to FFH and register AH set to one of the following physical error codes:

01H - Disk I/O Error : Permanent Error
02H - Read/Only Disk
04H - Invalid Drive : Drive Select Error
09H - Illegal ? in FCB

F_SFIRST

Find The First File That Matches The Specified FCB

Entry Parameters:
Register CL: 011H (17)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Directory Code
AH: Physical or Extended Error
BX: Same as AX

F_SFIRST scans the directory for a match with the referenced FCB. Two types of searches can be performed. For standard searches, the calling process initializes bytes 0 through 12 of the referenced FCB, with byte 0 specifying the drive directory to be searched, bytes 1 through 11 specifying the file or files to be searched for, and byte 12 specifying the extent. Byte 12 is usually set to 00H.

An ASCII question mark (63, or 03FH hexadecimal) in any of the bytes 1 through 12 matches all entries on the directory in the corresponding position. This facility, called ambiguous file reference, can be used to search for multiple files on the directory. When called in the standard mode, F_SFIRST scans for the first file entry in the specified directory that matches the FCB and belongs to the current user number.

F_SFIRST also initializes the F_SNEXT call. After the search call has located the first directory entry matching the referenced FCB, F_SNEXT can be called repeatedly to locate all remaining matching entries. In terms of execution sequence, however, the F_SNEXT call must follow either an F_SFIRST or F_SNEXT call with no other intervening BDOS file-access system calls.

If byte 0 of the referenced FCB is set to a question mark, F_SFIRST ignores the remainder of the referenced FCB and locates the first directory entry residing on the current default drive. All remaining directory entries can be located by making multiple F_SNEXT calls.

This type of search operation is not usually made by application programs, but it does provide complete flexibility to scan all directory entries. Note that this type of search operation must be performed to access a drive's directory label.

Upon return, F_SFIRST returns a directory code in register AL with the value 0 to 3 if the search is successful, or 0FFH if a matching directory entry is not found. Register AH is set to zero in both of these cases. For successful searches, the current DMA is also filled with the directory record containing the matching entry, and the relative starting position is (AL*32). The directory information can be extracted from the buffer at this position.

If the directory has been initialized for date and time stamping, then an FCB resides in every fourth directory entry, and successful directory codes are restricted to the values 0 to 2. For successful searches, if the matching directory record is an extent zero entry, and if an SFCB resides at offset 96 within the current DMA buffer, then the contents of (DMA Address + 96) = 021H, and the SFCB contains the time and date stamp information and password mode for the file. This information is located at the relative starting position of 97 + (AL * 10) within the current DMA in the following format:

0 - 3 Create or Access Date and Time Stamp Field
4 - 7 Update Date and Time Stamp Field
8 Password Mode Field

Refer to Section 2.8 for more information about SFCBs.

If a physical error is encountered, F_SFIRST performs different actions depending on the BDOS error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message identifying the error at the console and terminates the calling process. Otherwise, it returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical error codes:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error

F_SIZE

Compute The Size Of A Disk File

Entry Parameters:
- Register CL: 023H (35)
- DX: FCB Address - Offset
- DS: FCB Address - Segment

Returned Values:
- Register AL: Directory Code
- AH: Physical or Extended Error
- BX: Same as AX, Random Record Field of FCB Set

F_SIZE determines the **virtual** file size, which is the address of the record immediately following the end of the file. The virtual size of a file corresponds to the physical size if the file is written sequentially. If the file is written in random mode, gaps might exist in the allocation, and the file might contain fewer records than the indicated size. For example, if a single record with record number 262,143, the Concurrent maximum, is written to a file using F_WRITERAND, then the virtual size of the file is 262,144 records even though only one data block is actually allocated.

To compute file size, the calling process passes the address of an FCB with bytes R0, R1, and R2 present. F_SIZE sets the random record field of the FCB to the Random Record Number + 1 of the last record in the file. If the R2 byte is set to 04H, and R0 and R1 are both zero, then the file contains the maximum record count, 262,144.

A process can append data to the end of an existing file by calling F_SIZE to set the random record position to the end of file, and then performing a sequence of random writes.

Note: The file need not be open in order to use F_SIZE. However, if the file is open in Locked mode and it has been extended by the calling process, the file must be closed before calling F_SIZE. Otherwise, F_SIZE returns an incorrect file size. F_SIZE returns the correct size for files open in Unlocked mode and Read-Only mode.

Upon return, F_SIZE returns a 00H in register AL if the file specified by the referenced FCB is found, or a 0FFH in register AL if the file is not found. Register AH is set to 00H in both cases.

If a physical or extended error is encountered, F_SIZE performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the error and terminates the process. Otherwise, F_SIZE returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended error codes:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error
09H - Illegal ? in FCB

F_SNEXT

Find A Subsequent File That Matches the
Specified FCB Of A Previous F_SFIRST Or F_SNEXT

Entry Parameters:
Register CL: 012H (18)

Returned Values:
Register AL: Directory Code
AH: Physical or Extended Error
BX: Same as AX

F_SNEXT is identical to F_SFIRST except that the directory scan continues from the last entry that was matched. F_SNEXT returns a directory code in register AL, analogous to F_SFIRST.

Note: In execution sequence, an F_SNEXT call must follow either an F_SFIRST or another F_SNEXT with no other intervening BDOS file-access system calls.

F_TIMEDATE

Return File Date Stamps And Password Mode

Entry Parameters:
Register CL: 066H (102)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Directory Code
AH: Physical Error
BX: Same as AX

F_TIMEDATE returns the time and date stamp information and password mode for the specified file in byte 12 and bytes 24 through 31 of the specified FCB. The calling process passes the address of an FCB in which the drive, filename, and type fields have been defined.

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

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

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

byte 24 - 27 SFCB Create or Access time stamp field
byte 28 - 31 SFCB Update time stamp field

Upon return, F_TIMEDATE returns a directory code in register AL with the value 00H if the operation is successful, or 0FFH if the specified file is not found. Register AH is set to 00H in both of these cases.

If a physical or extended error is encountered, F_TIMEDATE performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the error and terminates the calling process. Otherwise, F_TIMEDATE returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical error codes:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error
09H - Illegal ? in FCB

F_TRUNCATE

Truncate File

Entry Parameters:
Register CL: 063H (99)
Register DX: FCB Address - Offset

Returned Values:
Register AL: Directory Code
Register AH: Physical or Extended Error
Register BX: Same as AX

F_TRUNCATE sets the last record of a file to the Random Record Number contained in the referenced FCB. The calling program passes the address of the FCB in register DX with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the filename and filetype, and bytes 33 through 35 (R0, R1, and R2) specifying the last record of the file. The last record number is a 24-bit value, stored with the least significant byte first (R0), the middle byte next (R1), and the high byte last (R2). This value can range from 0 to 262,143 (03FFFFH)

If the file specified by the referenced FCB is password-protected, the correct password must have been placed in the first eight bytes of the current DMA buffer, or have been previously established as the default password (see F_PASSWD).

Interface attribute F5' specifies whether an extended file lock is to be maintained after the F_TRUNCATE call, as shown below:

F5'= 0 Do not maintain an extended file lock (default)
F5'= 1 Maintain an extended file lock

If F5' is set and the referenced FCB specifies a file with an extended file lock, the calling process maintains the lock on the file. Otherwise, the file becomes available to other processes. Section 2.11 describes extended file locking in detail.

F_TRUNCATE requires that the Random Record Number field of the referenced FCB specify a value less than the current file size. In addition, if the file is sparse, the random record field must specify a region of the file where data exists.

A process can truncate a file that it currently has open if the file is opened in Locked mode, and the file has not been extended during the open session. However, the BDOS returns a checksum error if the process makes a subsequent reference to the file with a BDOS system call requiring an open FCB. A process cannot truncate files open in R/O or Unlocked mode.

Truncating an open file is not recommended. F_TRUNCATE truncates a file based on the file's state in the directory. If a process attempts to truncate at a region of the file that has been allocated in memory but has not been recorded in the directory, F_TRUNCATE returns an error. Even when successful, an open file truncate can adversely affect the performance of the calling process. For these reasons, you should close an open file before you truncate it.

After completion, F_TRUNCATE returns a directory code in register AL with the value 00H if the operation is successful or 0FFH if the file is not found or if the record number is invalid. In both cases register AH is set to 00H.

If a physical or extended error is encountered, F_TRUNCATE performs different actions depending on the BDOS error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the error and terminates the calling process. Otherwise, F_TRUNCATE returns to the calling program with register AL set to 0FFH and register AH set to one of the following physical or extended error codes:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
03H - Read/Only File
04H - Invalid Drive : drive select error
05H - File Currently Open
06H - Close Checksum Error
07H - Password Error
08H - File Already Exists
09H - Illegal ? in FCB
0AH - Open File Limit Exceeded
0BH - No Room in System Lock List

F_UNLOCK

Unlock Records In A Disk File

Entry Parameters:
- Register CL: 02BH (43)
- DX: FCB Address - Offset
- DS: FCB Address - Segment

Returned Values:
- Register AL: Error Code
- AH: Physical Error
- BX: Same as AX

F_UNLOCK unlocks one or more consecutive records previously locked by an F_LOCK call. F_UNLOCK is only supported for files open in Unlocked mode. If it is called for an file open in Locked or Read-Only mode, no unlocking action occurs but a successful result is returned. Record locking and unlocking is described in detail in Section 2.14.

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

If interface attribute F5' is set to 1, F_UNLOCK unlocks all locked records belonging to the calling process. The F_UNLOCK interface attribute definition is listed below:

- F5'= 0 Unlock records specified by Random Record Number and BDOS Multisector Count (default)
- F5'= 1 Unlock all locked records.

F_UNLOCK ignores the FCB Random Record field and the BDOS Multisector Count when F5' is set.

F_UNLOCK does not unlock a record that is currently locked by another process. However, F_UNLOCK does not return an error if a process attempts to do that. Thus, if the Multisector Count is greater than one, F_UNLOCK unlocks all records locked by the calling process, skipping those records locked by other processes.

Some F_UNLOCK requests require a new entry in the BDOS system Lock List. If there is insufficient space in the system Lock List to satisfy the F_UNLOCK request, or if the process record Lock List limit is exceeded, then F_UNLOCK does not unlocks any records and returns an error code to the calling process.

Upon return, F_UNLOCK sets register AL to 00H if the unlock operation was successful. Otherwise, register AL contains one of the following error codes:

01H - Reading unwritten data
03H - Cannot close current extent
04H - Seek to unwritten extent
06H - Random Record Number out of range
0AH - FCB Checksum Error
0CH - Process record Lock List limit exceeded
0DH - Invalid File ID
0EH - No room in system Lock List
0FFH - Physical error refer to register AH

F_UNLOCK returns error code 01H when it accesses a data block which has not been previously written.

F_UNLOCK returns error code 03H when it cannot close the current extent prior to moving to a new extent.

F_UNLOCK returns error code 04H when it accesses an extent that has not been created.

F_UNLOCK returns error code 06H when byte 35 (r2) for a list of the referenced FCB is greater than 3.

F_UNLOCK returns error code 0AH if the referenced FCB failed the FCB checksum test.

F_UNLOCK returns error code 0CH if performing the unlock request would require that the process consume more than the maximum allowed number of system Lock List entries.

F_UNLOCK returns error code 0DH when an invalid File ID is placed at the beginning of the current DMA.

F_UNLOCK returns error code 0EH when the system Lock List is full and performing the unlock request would require at least one new entry.

F_UNLOCK returns error code 0FFH if a physical error was encountered and the BDOS Error mode is one of the return modes (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the physical error and terminates the calling process. When F_UNLOCK returns a physical error to the calling process, it is identified by register AH as shown below:

01H - Disk I/O Error : permanent error
04H - Invalid Drive : drive select error

F_USERNUM

Set Or Return The Calling Process's Default User Number

Entry Parameters:
Register CL: 020H (32)
DL: 0FFH to GET User Number
User Number to SET

Returned Values:
Register AL: Current User Number if GET
BL: Same as AL

F_USERNUM can change or interrogate a process's current default user number. If register DL = 0FFH, then F_USERNUM returns the value of this user number in register AL. The value can range from 0 to 0FH. If register DL is not 0FFH, then F_USERNUM changes the default user number to the value in DL, modulo 010H (the high nibble of DL is masked off).

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

The operation of the Terminal Message Process (TMP) demonstrates this convention. When you enter a command, Concurrent creates a new process with the same user number as that of the TMP. If this new process changes its user number, the TMP is unaffected. Once the new process terminates, the TMP displays the same user number in its prompt that it displayed before you entered the command and the child process was created.

F_WRITE

Write Records Sequentially To A Disk File

Entry Parameters:
Register CL: 015H (21)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Error Code
AH: Physical Error
BX: Same as AX

F_WRITE writes 1 to 128, 128-byte data records beginning at the current DMA address into the file named by the specified FCB. The BDOS Multisector Count (see F_MULTISEC) determines the number of 128-byte records that are written. The default is one record. An F_OPEN or F_MAKE call must have previously activated the referenced FCB.

F_WRITE places the record into the file at the position indicated by the CR byte of the FCB, and then automatically increments the CR byte to the next record position. If the CR field overflows, F_WRITE automatically opens or creates the next logical extent and resets the CR field to 00H in preparation for the next write operation.

If F_WRITE is used to write to an existing file, then the newly written records overlay those already existing in the file. The calling process must set the CR field to 00H following an F_OPEN or F_MAKE call if the intent is to write sequentially from the beginning of the file.

F_WRITE makes an update date and time stamp for the file if the following conditions are met: the referenced drive has a directory label that requests update date and time stamping, and the file has not already been stamped for update by a previous F_MAKE or F_WRITE call.

Upon return, F_WRITE sets register AL to 00H if the write operation is successful. Otherwise, register AL contains an error code identifying the error as shown below:

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

F_WRITE returns error code 01H when it attempts to create a new extent that requires a new directory entry, and no available directory entries exist on the selected disk drive.

F_WRITE returns error code 02H when it attempts to allocate a new data block to the file, and no unallocated data blocks exist on the selected disk drive.

F_WRITE returns error code 08H if the calling process attempts to write to a record locked by another process, or a record locked by the calling process in shared mode. F_WRITE returns this error only for files open in Unlocked mode.

F_WRITE returns error code 09H if the FCB is invalidated by a previous F_CLOSE system call that returned an error.

F_WRITE returns error code 0AH if the referenced FCB fails the FCB checksum test.

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

F_WRITE returns error code 0FFH if a physical error was encountered and the BDOS is in Return Error mode or Return and Display Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the physical error and terminates the calling process. When F_WRITE returns a physical error to the calling process, it is identified by register AH as shown below:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
03H - Read/Only File or
      File Opened in Read/Only Mode or
      File password protected in Write mode
04H - Invalid Drive : drive select error

On all error returns except for physical error returns (AL = 255), F_WRITE sets register AH to the number of records successfully written before the error was encountered. This value can range from 0 to 127, depending on the current BDOS Multisector Count. It is always set to zero when the Multisector Count is equal to one.

F_WRITERAND

Write Random Records To A Disk File

Entry Parameters:
Register CL: 022H (34)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Error Code
AH: Physical Error
BX: Same as AX

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

In order to write to a file using F_WRITERAND, the calling process must first open the base extent, extent 0. This ensures that the FCB is properly initialized for subsequent random access operations. If the file is empty, the calling process must create the base extent with an F_MAKE call before calling F_WRITERAND. The base extent might or might not contain data, but it records the file in the directory so that it can be displayed by the DIR utility. If a process does not open extent 0 and allocates data to some other extent, the file is invisible to the DIR utility.

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

F_WRITERAND makes an update date and time stamp for the file if the following conditions are met: the referenced drive has a directory label that requests update date and time stamping, and the file has not already been stamped for update by a previous F_MAKE or F_WRITE call.

If the BDOS Multisector Count is greater than one (see F_MULTISEC), F_WRITERAND reads multiple consecutive records into memory beginning at the current DMA address. F_WRITERAND automatically increments the R0, R1,and R2 field of the FCB to write each record. However, it restores the FCB's Random Record Number to the first record's value upon return to the calling process.

Upon return, F_WRITERAND sets register AL to 00H if the write operation is successful. Otherwise, register AL contains one of the following error codes:

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

F_WRITERAND returns error code 02H when it attempts to allocate a new data block to the file. No unallocated data blocks exist on the selected disk drive.

F_WRITERAND returns error code 03H when it cannot close the current extent before moving to a new extent.

F_WRITERAND returns error code 05H when it attempts to create a new extent that requires a new directory entry and no available directory entries exist on the selected disk drive.

F_WRITERAND returns error code 06H when byte 35 (R2) of the referenced FCB is greater than 3.

F_WRITERAND returns error code 08H if the calling process attempts to write to a record locked by another process, or a record locked by the calling process in Shared mode. F_WRITERAND returns this error only for files open in Unlocked mode.

F_WRITERAND returns error code 0AH if the referenced FCB fails the FCB checksum test.

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

F_WRITERAND returns error code 0FFH if a physical error is encountered and the BDOS Error mode is in one of the return modes (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the physical error and terminates the calling process. When a physical error is returned to the calling process, it is identified by register AH as shown below:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
03H - Read/Only File or
File Opened in Read/Only Mode or
File password protected in Write mode
04H - Invalid Drive : drive select error

On all error returns, except for physical error returns (AL = 255), F_WRITERAND sets register AH to the number of records successfully written before the error was encountered. This value can range from 0 to 127 depending on the current BDOS Multisector Count. It is always set to zero when the Multisector Count is equal to one.

F_WRITEXFCB

Write Extended File Control Block Of A Disk File

Entry Parameters:
Register CL: 067H (103)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Directory Code
AH: Physical or Extended Error
BX: Same as AX

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

FCB byte 12 (EX) XFCB password mode

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

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

Note: F_WRITEXFCB does not create or update an XFCB if the XFCB specifies a file open by another process. However, a process can update or create an XFCB for a file that it has open in Locked mode.

Upon return, F_WRITEXFCB returns a directory code in register AL with the value 00H if the XFCB create or update was successful. F_WRITEXFCB returns 0FFH in register AL if no directory label existed on the specified drive, or the file specified in the FCB was not found, or no space existed in the directory to create an XFCB, or if the drive is not password enabled. F_WRITEXFCB also returns 0FFH if passwords are not enabled by the specified drive's directory label. Register AH is set to 00H in all of these cases.

If a physical or extended error is encountered, F_WRITEXFCB performs different actions depending on the BDOS Error mode (see F_ERRMODE). If the BDOS is in the default Error mode, Concurrent displays a message at the console identifying the error and terminates the calling process. Otherwise, F_WRITEXFCB returns to the calling process with register AL set to 0FFH and register AH set to one of the following physical or extended error codes:

01H - Disk I/O Error : permanent error
02H - Read/Only Disk
04H - Invalid Drive : drive select error
05H - File open by another process,
or open in Read-Only or Unlocked mode
07H - Password Error
09H - Illegal ? in FCB

F_WRITEZF
Write A Random Record To A Disk File
And Prefill New Data Blocks With Zeros

Entry Parameters:
Register CL: 028H (40)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AL: Error Code
AH: Physical Error
BX: Same as AX

F_WRITEZF is similar to F_WRITERAND, with the exception that it fills a previously unallocated data block with zeros (00H) before writing the record. If F_WRITEZF has been used to create a file, records accessed by F_READRAND that contain all zeros identify unwritten random records. Unwritten random records in allocated data blocks of files created using F_WRITERAND contain uninitialized data.

L_ATTACH

Attach The Default List Device To The Calling Process

Entry Parameters:
Register CL: 09EH (158)

L_ATTACH attaches the default list device of the calling process. If the list device is already attached to some other process, the calling process relinquishes the CPU until the other process detaches from the list device. When the list device becomes free, and the calling process is the highest priority process waiting for the list device, the attach operation occurs.

L_CATTACH

Conditionally Attach To The Default List Device

Entry Parameters:
Register CL: 0A1H (161)

Returned Values:
Register AX: 0 if attach 'OK', 0FFFFH on failure
BX: Same as AX
CX: Error Code

L_CATTACH attaches the default list device of the calling process only if the list device is currently available.

If the list device is currently attached to another process, L_CATTACH returns a value of 0FFH, indicating that the list device could not be attached. L_CATTACH returns a value of 00H to indicate that either the list device is already attached to the process, or that it was unattached, and a successful attach operation was made.

Table 6-5 contains the list of error codes returned in CX.

L_DETACH

Detach The Default List Device From The Calling Process

Entry Parameters:
Register CL: 09FH (159)

Returned Values:
Register AX: 0 if detach 'OK', 0FFFFH on failure
BX: Same as AX

L_DETACH detaches the default list device of the calling process. If the list device is not currently attached, no action takes place.

L_GET

Return The Calling Process's Default List Device

Entry Parameters:
Register CL: 0A4H (164)

Returned Values:
Register AL: List Device Number
BL: Same as AL

L_GET returns the default list device number of the calling process.

L_SET

Set The Calling Process's Default List Device

Entry Parameters:
Register CL: 0A0H (160)
DL: List Device Number

Returned Values:
Register CX: Error Code

L_SET sets the default list device for the calling process.

Table 6-5 contains the list of error codes returned in CX.

L_WRITE

Write A Character To The Default List Device

Entry Parameters:
Register CL: 05H (5)
DL: Character

L_WRITE writes the specified character to the default list device of the calling process. Before writing the character, Concurrent calls L_ATTACH to verify that the calling process owns its default list device.

L_WRITEBLK

Send Specified Character String to Default List Device

Entry Parameters:
Register CL: 070H (112)
Register DX: CHCB Address

L_WRITEBLK sends the character string specified in the **Character Control Block** (CHCB) and addressed in register pair DX to the logical list device, LST:. The CHCB format is:

bytes 0 - 1	Offset of character string
bytes 2 - 3	Segment of character string
bytes 4 - 5	Length of character string to print

Memory Call Data Structures

There are two classes of Memory system calls in Concurrent DOS 86. The first class supports the MP/M-86 memory allocation scheme and contains two calls: M_ALLOC and M_FREE.

The second class supports the CP/M-86 memory allocation scheme and contains six calls: MC_ABS, MC_ALLFREE, MC_ALLOC, MC_ALLOCABS, MC_FREE, and MC_MAX.

Note: The CP/M-86 memory calls are also supported under MP/M-86.

Many of the Memory calls use the **Memory Control Block** (MCB) or the **Memory Parameter Block** (MPB) to pass parameters to and from Concurrent.

Figure 6-7 shows the Memory Control Block, Table 6-14 defines its fields, and Listing 6-1 shows the programming equates for this data structure.

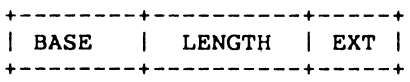


Figure 6-7. MCB - Memory Control Block

Table 6-14. MCB Field Definitions

Field	Definition
BASE	The Segment Address of the beginning of the specified memory segment.
LENGTH	Length of the Memory Segment in paragraphs. The LENGTH field is set to the number of paragraphs wanted.
EXT	The EXT field is unused but must be available.

Listing 6-1. Memory Control Block Definition

```
;************************************************
;*
;*      Memory Control Block Definition
;*
;************************************************
mcb_base        equ     word ptr 0
mcb_length      equ     word ptr mcb_base + word
mcb_ext         equ     byte ptr mcb_length + word

mcb_len         equ     mcb_ext + byte
;
```

Figure 6-8 shows the Memory Control Block, Table 6-15 defines its fields, and Listing 6-2 shows the programming equates for this data structure.

Figure 6-8. MPB - Memory Parameter Block

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

Table 6-15. MPB Field Definitions

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

Listing 6-2. Memory Parameter Block Definition

```
;*****************************************************
;*
;*      Memory Parameter Block Definition
;*
;*****************************************************
mpb_start       equ     word ptr 0
mpb_min         equ     word ptr mpb_start + word
mpb_max         equ     word ptr mpb_min + word
mpb_pdadr       equ     word ptr mpb_max + word
mpb_flags       equ     word ptr mpb_pdadr + word

mpb_len         equ     mpb_flags + word

        ; mpb_flags definition

mf_load         equ     00001h
mf_share        equ     00002h
mf_code         equ     00004h
;
```

EMM Data Structures

Expanded Memory Management requires two additional memory call data structures. These are the **Memory Window Descriptor** and the **Memory Page Allocation Descriptor**.

The XIOS maintains one Memory Window Descriptor (MWD) for each logical address window of paged memory in the system. MWD's are arranged as a linked list pointed to by the **Memory Window Descriptor Root** (MWDR) located at offset 98H in the System Data Area (see SYSDAT).

The Memory Page Allocation Descriptor (MPAD) is a dynamic structure created by MEM whenever paged memory is allocated to a process. MPADs are arranged as a linked list pointed to by the **Memory Page Allocation Root** (P_MPAR) field in the Process Descriptor of the process owning the unit of paged memory. The P_MPAR field is located at offset 34H in the Process Descriptor (see P_CREATE).

See the System Guide for additional information about these data structures.

M_ALLOC

Allocate A Memory Segment

Entry Parameters:
- Register CL: 080H or 081H (128,129)
- DX: MPB Address Offset
- DS: MPB Address Segment
- MPB filled in

Returned Values:
- Register AX: 0 on success, 0FFFFH on failure
- BX: Same as AX
- CX: Error Code
- MPB_start filled in

M_ALLOC allows a program to allocate extra memory. A successful allocation allocates a contiguous memory segment whose length is at least the MIN and no more than the MAX number of paragraphs specified in the MPB.

The START field of the MPB is modified to be the starting paragraph of the memory segment. The MIN and MAX fields are modified to be the length of the memory segment in paragraphs. Memory Segments can be explicitly released with M_FREE; Concurrent also releases all memory owned by a process at termination.

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

Table 6-5 contains the list of error codes returned in CX.

M_FREE

Free A Memory Segment

Entry Parameters:
Register CL: 082H (130)
DX: MFPB Address - Offset
DS: MFPB Address - Segment

Returned Values:
Register AX: 0 on success, 0FFFFH on failure
BX: Same as AX
CX: Error Code

The calling process passes the address of a **Memory Free Parameter Block** (MFPB) as shown in the Figure 6-9.

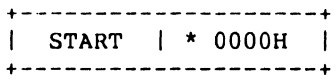


Figure 6-9. MFPB - M_FREE Parameter Block

M_FREE releases memory starting at the START paragraph to the end of a single previously allocated segment that contains the START paragraph. If the START paragraph is the same as that returned in the MPB of a memory allocation call, then M_FREE releases the whole memory segment. The * 0000H field must be initialized to zero.

Table 6-5 contains the list of error codes returned in CX.

MC_ABSALLOC

Allocate A Memory Segment At A Specified Address

Entry Parameters:
- Register CL: 038H (56)
- DX: MCB Address - Offset
- DS: MCB Address - Segment

Returned Values:
- Register AL: 0 on success, 0FFH on failure
- BL: Same as AL
- CX: Error Code

MC_ABSALLOC allocates a memory area starting at the address specified by the BASE field. The memory area's length is specified by the LENGTH field of the MCB. Upon return, register AL contains a 00H if the request was successful, and a 0FFH if the memory could not be allocated. If the calling process already owns the requested memory, no error is returned. This assures compatibility with CP/M-86.

Table 6-5 contains the list of error codes returned in CX.

MC_ABSMAX

Allocate Maximum Memory Available At A Specified Address

Entry Parameters:
- Register CL: 036H (54)
- DX: MCB Address - Offset
- DS: MCB Address - Segment
  MCB_base filled in, MCB_length set to max number of paragraphs wanted

Returned Values:
- Register AL: 0 on success, 0FFH on failure
- BL: Same as AL
- CX: Error Code
  MCB_length set to actual number of paragraphs allocated

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

Undr CP/M-86, this call does not allocate memory, but under Concurrent CP/M, it does because other processes are competing for common memory. For compatibility with CP/M-86, MC_ABSALLOC (system call 56) does not return an error if there is a memory segment allocated at the absolute address.

Table 6-5 contains the list of error codes returned in CX.

MC_ALLFREE

Free All Memory Owned By The Calling Process

Entry Parameters:
Register CL: 03AH: (58)

Under Concurrent, MC_ALLFREE releases all of the calling process's memory except the User Data Area (UDA). It is useful for system processes and for subprocesses that share the memory of another process.

Note: MC_ALLFREE should not be used by processes running programs loaded into the Transient Program Areas (TPA).

MC_ALLOC

Allocate A Memory Segment

Entry Parameters:
- Register CL: 037H (55)
- DX: MCB Address - Offset
- DS: MCB Address - Segment
- MCB_length filled in

Returned Values:
- Register AL: 0 on success, 0FFH on failure
- BL: Same as AL
- CX: Error Code
- MCB_base filled in

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

Table 6-5 contains the list of error codes returned in CX.

MC_FREE

Free A Specified Memory Segment

Entry Parameters:
- Register CL: 039H (57)
- DX: MCB Address - Offset
- DS: MCB Address - Segment
  MCB_base, MCB_ext filled in

Returned Values:
- Register AL: 0 if successful, 0FFH on failure
- BL: Same as AL
- CX: Error Code

MC_FREE releases memory areas allocated to the process. The value of the EXT field of the MCB controls the free operation. If EXT = 0FFH, then MC_FREE releases all memory areas allocated by the calling process. If the EXT field is 00H, then MC_FREE releases the memory area beginning at the specified BASE and ending at the end of the previously allocated memory segment.

Table 6-5 contains the list of error codes returned in CX.

MC_MAX

Allocate Maximum Memory Available

Entry Parameters:

Register CL:	035H (53)
DX:	MCB Address - Offset
DS:	MCB Address - Segment
	(MCB_length contains maximum number of paragraphs wanted)

Returned Values:

Register AL:	0 on success, 0FFH on failure
BL:	Same as AL
CX:	Error Code
	(MCB_base filled in, MCB_length set to actual number of paragraphs allocated)

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

Under CP/M-86, this call does not allocate memory, but under Concurrent it does because other processes are competing for common memory. For compatibility with CP/M-86, MC_ABSALLOC (system call 56) does not return an error if there is a memory segment allocated at the absolute address.

Table 6-5 contains the list of error codes returned in CX.

P_ABORT

Terminate A Process By Name Or PD Address

Entry Parameters:
- Register CL: 09DH (157)
- DX: APB Address - Offset
- DS: APB Address - Segment
  APB filled in

Returned Values:
- Register AX: 0 on success, 0FFH on failure
- BX: Same as AX
- CX: Error Code

P_ABORT terminates a specified process by passing the address of a data structure called an **Abort Parameter Block** (APB). Figure 6-10 shows the format of the APB and Table 6-16 lists the APB field definitions.

The process name and console can be omitted if the Process Descriptor address is filled in. Otherwise, the Process Descriptor address field should be a 00H and the process name and console must be specified. In either case, the calling process must supply the termination code, which is the same parameter passed to the P_TERM call.

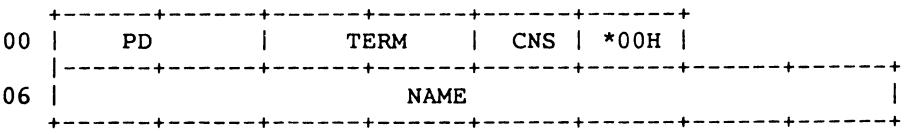


Figure 6-10. APB - Abort Parameter Block

Table 6-16. APB Field Definitions

Field	Definition
PD	Process Descriptor offset of the process to be terminated. If this field is zero, a match is attempted with the NAME and CNS fields to find the process. If this field is nonzero, the NAME and CNS fields are ignored.
TERM	Termination Code. This field corresponds to the termination code of the P_TERM call. If the low-order byte of TERM is 0FFH, P_ABORT can abort a specified system process; if the termination code is not 0FFH, P_ABORT can only abort a user process. (A system process is identified by the SYS flag in the Process Descriptor's FLAG field.)
*00H	This field is reserved for future use and must be set to zero.
CNS	Default console of process to be aborted. If the PD field is 0, P_ABORT scans the Thread List for a PD with the same NAME and CNS fields as specified in the APB. P_ABORT only aborts the first process that it finds. Subsequent calls must be made to abort all processes with the same NAME and CNS.
NAME	Name of the process to be aborted. Combined with the CNS field, the NAME field is used to find the process to be aborted. This is only used if the PD field is 0.

Table 6-5 contains the list of error codes returned in CX.

P_CHAIN

Load, Initialize And Jump To Specified Program

Entry Parameters:
Register CL: 02FH (47)
DMA Buffer: Command Line

Returned Values:
Register AX: 0FFFFH - Could not find Command

P_CHAIN chains from one program to the next without user intervention. Although P_CHAIN requires no passed parameter, the calling process must place a command line terminated by a 0 byte in the default DMA buffer.

P_CHAIN releases the memory of the calling process before executing the command. The command is processed in the same manner as the P_CLI call. If the command warrants the loading of a CMD file and the memory released is large enough for the new program, Concurrent loads the new program into the same memory area as the old program. The new program is run by the same process that ran the old program. The name of the process is changed to reflect the new program being run.

Parameter passing between the old and new programs is accomplished through the use of disk files, queues, or the command line. The command line is parsed and placed in the Base Page of the new program as described in P_CLI.

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

P_CLI

Interpret And Execute Command Line

Entry Parameters:
Register CL: 096H (150)
DX: CLBUF Address - Offset
DS: CLBUF Address - Segment

Returned Values:
Register AX: 0 on success, 0FFFFH on error
CX: Error Code

P_CLI obtains an ASCII command from the **Command Line Buffer** (CLBUF) and then executes it. Figure 6-11 shows the Command Line Buffer format and Table 6-17 lists the CLBUF field definitions.

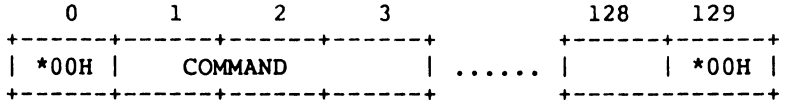


Figure 6-11. CLI Command Line Buffer

Table 6-17. Command Line Buffer Field Definitions

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

If the calling process is attached to its default virtual console, P_CLI assigns the virtual console to either the newly created process, or to the Resident System Process (RSP) that acts on the command. The calling process must reattach to its default virtual console before accessing it.

P_CLI calls F_PARSE to parse the command line. If an error occurs in F_PARSE, P_CLI returns to the calling process with the error code set by F_PARSE.

If there is no disk specification for the command, P_CLI tries to open a system queue with the same name as the command. If the open operation is successful, and the queue is an RSP-type queue, P_CLI then writes the command tail to the RSP queue. If the queue is full, P_CLI returns an error code to the calling process.

P_CLI also attempts to assign the calling process's virtual console to a process with the same name as the RSP queue. If the RSP queue cannot be found, the P_CLI assumes the command is on disk and continues.

P_CLI opens a file with the filename being the command and the filetype being CMD. If the command has an explicit disk specification, and the F_OPEN call fails, P_CLI returns an error code to the calling process. If there is no disk specification with the command, P_CLI attempts to open the command file on the system disk. If the F_OPEN call succeeds, P_CLI checks the file to verify the SYSTEM attribute is on. This search order is discussed in the User's Guide. If this second F_OPEN fails or if the DIR attribute is on, P_CLI returns an error code to the calling process.

Once P_CLI succeeds in opening the command file, it calls P_LOAD to find and then load the file into an appropriate memory space. If P_LOAD encounters any errors, P_CLI returns to the calling process with the error code set by P_LOAD.

A successful load operation establishes the command file in memory with its Base Page partially initialized. P_CLI then continues parsing the command tail to set up the Base Page values from 050h to 0FFH.

P_CLI initializes an unused Process Descriptor from the internal PD table, a UDA (expanded UDA if 8087 processing is required) and a 96-byte stack area. The UDA and stack are dynamically allocated from memory. P_CLI then calls P_CREATE.

If P_CLI encounters an error in any of these steps, it releases all memory segments allocated for the new command, as well as the Process Descriptor, and then returns with the appropriate error code set.

Once P_CREATE returns successfully, P_CLI assigns the calling process's default virtual console to the new process and then returns.

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

Table 6-5 contains the list of error codes returned in CX.

P_CREATE

Create A Process

Entry Parameters:
- Register CL: 090H (144)
- DX: PD Address - Offset
- DS: PD Address - Segment
- PD filled in

Returned Values:
- Register AX: 0 on success, 0FFFFH on failure
- BX: Same as AX
- CX: Error Code

P_CREATE creates a subprocess within a process's own memory area. The child process shares all memory owned by the calling process at the time of the P_CREATE call. The calling process passes the address of a **Process Descriptor** (PD). Figure 6-12 shows the Process Descriptor format and Table 6-18 lists the PD field definitions.

Process Descriptors, as well as Queue Descriptors and Queue Buffers, are required to be within the System Data Segment because they are linked together on various system lists or are used by more than one process. Because of this, they cannot be in the Transient Process Area (TPA), where they are unprotected, so Concurrent copies all Process Descriptors into an internal PD table. P_CREATE returns an error code if there are no more unused PDs in the table.

A single P_CREATE call can create more than one process if the PD's LINK field is nonzero. In this case, it is assumed to point to another PD within the same Data Segment. After it creates the first process, P_CREATE checks the LINK field and if non-zero, it follows the linked list to create multiple processes.

WARNING! P_CREATE does not check the validity of the PD addresses passed by the calling process. If there is no hardware memory protection on the system, passing an invalid PD address can cause Concurrent to halt indefinitely requiring a reboot.

Table 6-5 contains the list of error codes returned in CX.

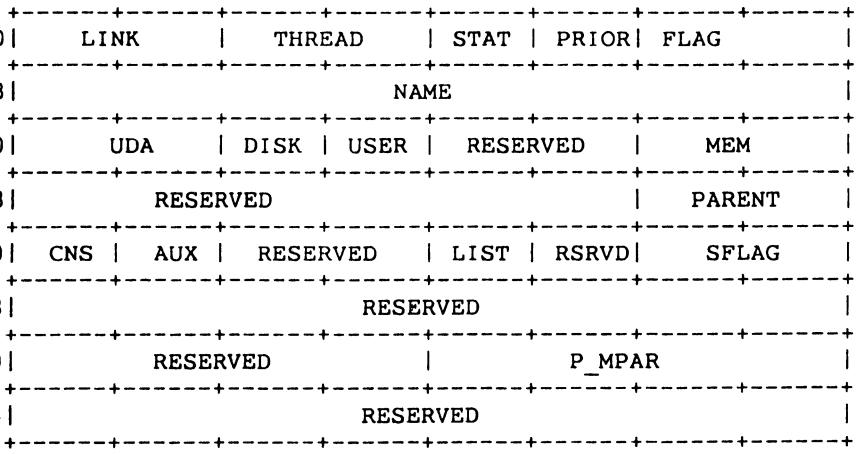


Figure 6-12. PD - Process Descriptor

Table 6-18. PD Field Definitions

Field	Definition	
LINK	Link field for insertion on current system list. If LINK's initial value is nonzero, it is assumed to point to another PD. Concurrent uses the LINK field to create more than one process with a single Create Process call.	
THREAD	Link field for insertion on Thread List. Initialized to zero (0).	
STAT	Current Process activity. Initialized to zero (0). Activity codes are listed below:	
	00 RUN	The process is ready to run. The STAT field is always in this state when a process is examining its own Process Descriptor. The PD is on the Ready List. The currently running process is always at the head of Ready List.
	01 POLL	The process is polling a device. The PD is on the Poll List.
	02 DELAY	The process is delaying for a specified number of system ticks. The PD is on the Delay List.
	06 Read Queue	The process is waiting to read a message from a system queue that is empty. The PD is on the Read Queue List, whose root is in the Queue Descriptor of the system queue involved.
	07 Write Queue	The process is waiting to write a message to a system queue whose buffer is full. The PD is on the Write Queue List, whose root is in the Queue Descriptor of the system queue involved.
	08 FLAGWAIT	The process is waiting for a system flag to be set. The PD is in the flag table entry of the flag it is waiting for.

Table 6-18. (Cont'd)

Field	Definition	
	09 CIOWAIT	The process is waiting to attach to a character I/O device (console or list) while another process owns it. The PD is on CQUEUE list whose root is in the Character Control Block of the device in question.
PRIOR	Current priority. Used to determine process scheduling. Typical user programs run at a priority of 200. 0 is the best priority, and 255 is the worst priority. The following list of priorities is used by most Concurrent systems. User processes priorities should be from 200-254.	
	1	Initialization Process
	2 - 31	Interrupt Handlers
	32 - 63	System Processes
	64 - 189	Undefined
	191 - 197	Undefined
	198	Terminal Message Process
	199	Undefined
	200	Default Priority For Transients
	201 - 254	User Processes
	255	Idle Process
FLAG	Bit field of flags determining run-time characteristics of a process. Initialize as needed. All undocumented flags are used internally or are reserved for future use.	
	001H SYS	System Process. Has privileged access to various features of Concurrent. This process can only be terminated if the termination code is 0FFH. This process can access restricted system queues. Turn flag off if the calling process is not a system process.
	002H KEEP	This process cannot be terminated. Turn flag off if the calling process is not a system process.
	004H KERNEL	This process resides within Concurrent. Turn flag off if the PD is not within Concurrent.

Table 6-18. (Cont'd)

Field	Definition	
	010H TABLE	This PD is copied into the PD from the PD table. When this process terminates, the PD is recycled into the PD table.
	8000H 8087	This process is an 8087-running process.
NAME	Process Name. Eight bytes, with all eight bits of each byte used for matching process names.	
UDA	Segment address of this process's User Data Area. The UDA can be anywhere in memory but must be on a paragraph boundary. Initialized to be the number of paragraphs from the beginning of the calling process's Data Segment. The UDA contains process information that is not needed between processes. It also contains the System Stack of each process. See Figure 6-13 and Table 6-19 for more detailed information about the UDA.	
DISK	Current default disk	
USER	Current default user number	
RESERVED	Reserved for internal use. Must be initialized to zero (0).	
MEM	Root of linked list of Memory Segment Descriptors that are owned by this process. Initialized to zero, except for reentrant or shared code RSPs.	
RESERVED	Reserved for internal use. Must be initialized to zero (0).	
PARENT	Process that created this process. P_CREATE sets this value at process creation. The parent field is set to zero if the parent terminates before the child.	
CNS	Current default console's number. Initialized to be the default console number.	
AUX	This field contains the current default auxiliary device number. Must be initialized to zero (0H) before calling P_CREATE.	
RESERVED	Reserved for internal use. Must be initialized to zero (0).	
LIST	Current default list device. Initialized to be the default list device number.	
RESERVED	Reserved for internal use. Must be initialized to zero (0).	

Table 6-18. (Cont'd)

Field	Definition
SFLAG	Second Flag. Bit field of flags determining run-time characteristics of a process. Initialize as needed. All undocumented flags are used internally or are reserved for future use and should not be set.
	Bit 0 — When bit 0 is set, Concurrent suspends this process whenever it is switched to the background, and runs it only when it is switched to the foreground.
	Bit 1 — Bit 1 is reserved for internal system use.
	Bit 2 — Concurrent sets bit 2 when this process is running in the foreground. This bit should be clear when the Process Descriptor is initialized.
	Bit 3 — When bit 3 is set, Concurrent does not use the Default System Disk when loading programs for this process.
	Bit 4 — Concurrent sets bit 4 when a user enters CTRL-C. Clear this bit to check for CTRL-C input.
	Bit 5 — If bit 5 is set, Concurrent resets the disk system when the user types CTRL-C. If bit 5 is clear, Concurrent interprets CTRL-C as process termination.
	Bit 6 — When bit 6 is set, Concurrent allows this process to access records locked by another process. This flag is included for MP/M-style record locking.
RESERVED	Reserved for internal use. Must be initialized to zero (0).
P_MPAR	Memory Page Allocation Root. Root of linked list of Memory Page Allocation Descriptors for paged memory allocated to calling process.
RESERVED	Reserved for internal use. Must be initialized to zero (0).

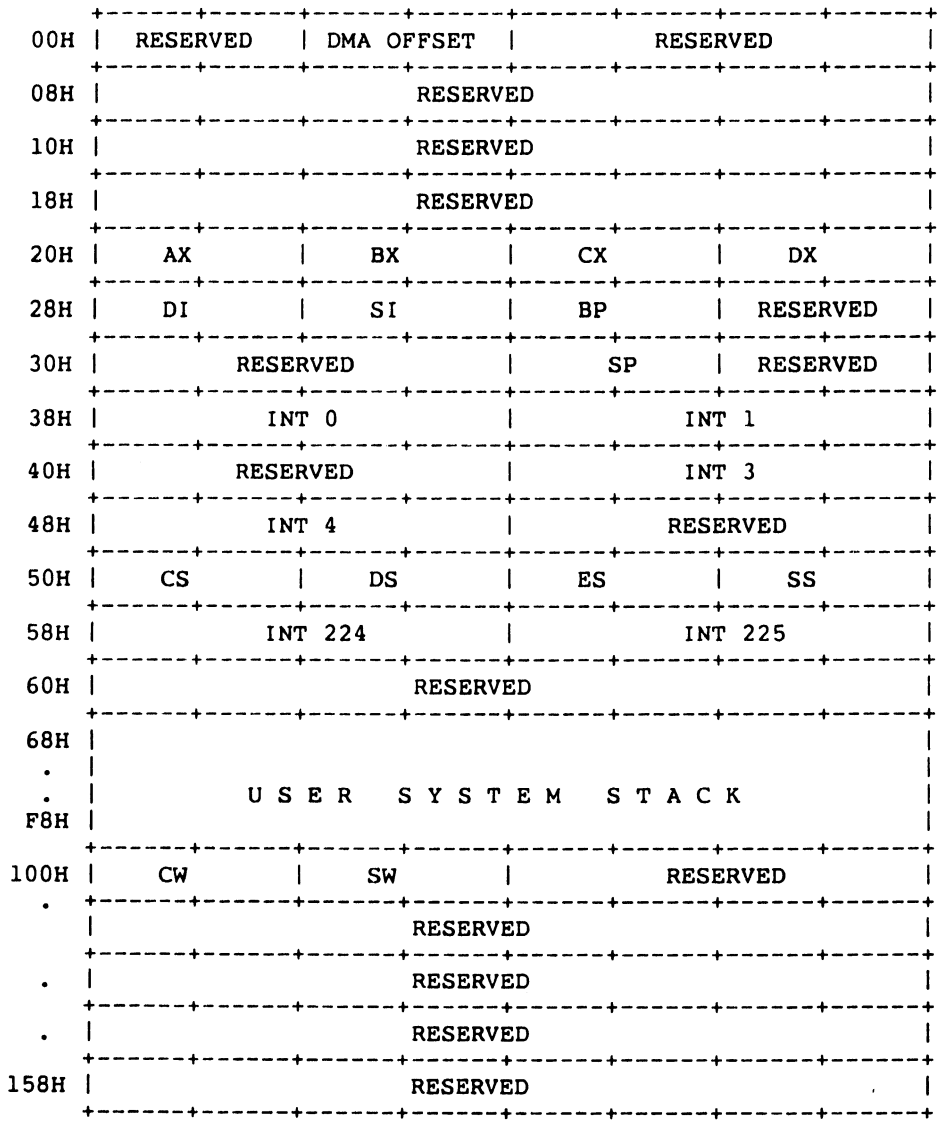


Figure 6-13. UDA - User Data Area

Note: The UDA length is 256 bytes, and it must begin on a paragraph boundary. If the optional 96-byte 8087 processing extension is used, the length is 352 bytes.

Table 6-19. UDA Field Definition

Field	Definition
DMA OFFSET	The initial DMA offset for the new process. The segment address of the DMA is assumed to be the same as the initial Data Segment (see DS below).
AX - BP	The initial register values for the new process. These are typically set to zero.
SP	The initial stack pointer for the new process. The stack pointer is relative to the initial Stack Segment (see SS below). The new process's initial stack must be initialized with the offset of the first instruction it is to execute. The word that the stack pointer points to is the initial instruction pointer (IP). Two words must follow the initial IP, which is filled in with the initial Code Segment (see CS below) and the initial flags. The initial flags are set to 0200H, which means that interrupts are on, and all other flags are off. Concurrent starts a new process by executing an Interrupt Return instruction with the initial stack. **Note:** This stack area is distinct from the User System Stack at the end of the UDA.
INT 0 - INT 4	The initial interrupt vectors for the first five interrupt types can be set by filling in these fields. The first word of each field is the Instruction Pointer (IP), and the second word is the Code Segment (CS) for a list of the interrupt routines that service these interrupts. Those fields that are zero are initialized to be the same as the calling processes interrupt vectors. These fields are typically initialized to be 0.
CS,DS,ES,SS	The initial segment addresses for the new process are taken from these fields. Those fields that are zero are initialized to be the same as the calling process's Data Segment.

Table 6-19. (Cont'd)

Field	Definition
INT 224,225	Interrupts 224 and 225 are used to communicate with Concurrent by typical programs. These interrupt vectors are initialized to be the same as the calling process if these values are zero. The ability to change these values allows a run-time system to intercept Concurrent calls that its children make. The suggested protocol is to keep INT 225 pointing to the Concurrent entry point and changing INT 224 to point to an internal routine. When a child process does an INT 224, the internal routine can filter calls to Concurrent using INT 225 for the actual Concurrent call.
RESERVED	These fields are used internally and must be initialized to zero.
USER SYSTEM STACK	This is the stack area used by the process when it is in the operating system. The SP variable in the UDA should not point to this area.
CW	Control Word for 8087 processing. RSPs (and other processes bypassing the P_CLI call) must set this word to 03FFH before system generation.
SW	Status Word for 8087 processing. RSPs (and other processes bypassing the P_CLI call) must set this word to 0000H before system generation.

P_DELAY

Delay For Specified Number Of System Ticks

Entry Parameters:
Register CL: 08DH (141)
DX: Number of System Ticks

P_DELAY delays execution of the calling process for a specific time interval, thus allowing other processes to use the CPU resource while the calling process waits. P_DELAY avoids the necessity of programmed delay loops.

The calling process specifies the delay interval as a number of system ticks. The length of the system tick varies among installations. A typical system tick is 60Hz (16.67 milliseconds). In Europe, it is likely to be 50Hz (20 milliseconds). The exact length can be obtained by reading the TICKSPERSEC value from the System Data Segment (see S_SYSDAT).

There is up to one tick of uncertainty in the exact amount of time delayed because P_DELAY is called asynchronously from the actual time base. P_DELAY is guaranteed to delay the calling process at least the number of ticks specified. However, when the calling process is rescheduled to run, it might wait quite a bit longer if there are higher priority processes waiting to run.

P_DELAY is useful for programs that need to wait specific amounts of time for I/O events to occur. Under these conditions, the calling process usually has a very high priority level. If a process with a high priority calls P_DELAY the actual delay is typically within a system tick of the amount of time wanted.

P_DISPATCH

Call Dispatcher

Entry Parameters:
Register CL: 08EH (142)

P_DISPATCH forces a reschedule of processes that are waiting to run. Normally, dispatches occur at every system tick interrupt (usually 60 times a second), and whenever a process releases a system resource.

Dispatching also occurs whenever a process needs a system resource that is not currently available. A CPU-bound process runs for no more than one system tick before a dispatch is forced. The dispatch occurs at the next system tick.

Concurrent's Dispatcher is priority driven, with round-robin scheduling of equivalent-priority processes. When a process calls P_DISPATCH, it is rescheduled, so that processes with higher or equivalent priorities are given the CPU before the calling process obtains it again. The calling process regains control of the CPU resource when it becomes the highest priority process again.

P_LOAD

Load a CMD type file into Memory

Entry Parameters:
Register CL: 03BH (59)
DX: FCB Address - Offset
DS: FCB Address - Segment

Returned Values:
Register AX: Base Page Address, 0FFFFH on error
BX: Same as AX
CX: Error Code

P_LOAD loads a disk CMD-type file into memory. Upon entry, register DX contains the offset, relative to DS, of a successfully opened FCB that specifies the CMD file to load. Upon return, register AX has the value 0FFFFH if the program load failed. Otherwise, AX contains the paragraph address of the Base Page belonging to the loaded program. The paragraph address and length of each group loaded from the CMD file, is found in the Base Page. See Sections 3.2 and 3.3.

Before calling P_LOAD, the calling process must establish the DMA address of where the CMD file is to be loaded. This is accomplished with F_DMASEG and F_DMAOFF.

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

Table 6-5 contains the list of error codes returned in CX.

P_PDADR

Return The Address Of The Calling Process's Process Descriptor

Entry Parameters:
Register CL: 09CH (156)

Returned Values:
Register AX: PD Address - Offset
BX: Same as AX
ES: PD Address - Segment

P_PDADR obtains the address of the calling process's Process Descriptor. See P_CREATE for a description the Process Descriptor format.

P_PRIORITY

Set The Priority Of The Calling Process

Entry Parameters:
Register CL: 091H (145)
DL: Priority

P_PRIORITY sets the priority of the calling process to the specified value. P_PRIORITY is useful when a process needs to have a high priority during an initialization phase, but afterwards can run at a lower priority.

The highest priority is 00H, while the lowest priority is 0FFH. Transient processes are initialized to run at C8H (200 decimal) by P_CLI.

P_RPL

Resident Procedure Library

Entry Parameters:
- Register CL: 097H (151)
- DX: CPB Address - Offset
- DS: CPB Address - Segment

Returned Values:
- Register AX: 01H if RPL not found, RPL return parameter
- BX: same as AX
- CX: Error Code
- ES: RPL return segment if addr

P_RPL permits a process to make system call from an optional **Resident Procedure Library** (RPL). The calling process passes the address of **Call Parameter Block** (CPB). Figure 6-14 shows the Call Parameter Block format, and Table 6-20 lists the CPB field definitions.

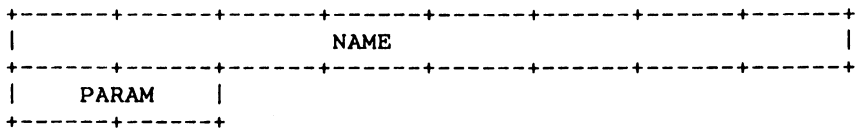


Figure 6-14. CPB - Call Parameter Block

Table 6-20. CPB Field Definitions

Field	Definition
NAME	Name of Resident Procedure, eight ASCII characters.
PARAM	Parameter to send to the Resident Procedure. P_RPL first opens a system queue with the specified name. If the Q_OPEN call succeeds, P_RPL verifies that it is an RPL-type queue. If either the Q_OPEN fails, or if it is not an RPL-type queue, P_RPL returns to the calling process with an error code.

P_RPL reads a message from the queue containing the address of the specified system call. It then places the PARAM field of the CPB in register DX, and places the calling process's Data Segment address in register DS. P_RPL performs a Far Call instruction to the address it obtains from the queue message. Upon return from the RPL, P_RPL copies the BX register to the AX register and then returns to the calling process.

Note: P_RPL does not write the address of the Resident Procedure back to the queue; the Resident Procedure itself must do this. If the Resident Procedure is to be reentrant, it must write the message into the queue upon entry. If it is to be serially reusable, the procedure must write the message just before returning.

Table 6-5 contains the list of error codes returned in CX.

P_TERM

Terminate Calling Process

Entry Parameters:
Register CL: 08FH (143)
DL: Termination Code

Returned Values:
Register AX: 0 on success, 0FFFFH on failure
BX: Same as AX

P_TERM terminates the calling process. If the Termination Code is not 0FFH, P_TERM can only terminate a user process; if the Termination Code is 0FFH, P_TERM can terminate the calling process even though the process's SYSTEM flag is on. P_TERM cannot terminate a process with the KEEP flag on.

If the termination is successful, P_TERM releases the Mutual Exclusion queues owned by the process. It also releases all memory segments owned by the process, and returns the Process Descriptor to the PD table.

A process can own one or more of the following resources: memory segments, consoles, printers, Mutual Exclusion messages, and system Lock List entries that record open files and locked records. When a process terminates and releases its resources, these resources become available to other processes on the system. For example, if a terminating process releases a system console, the console is usually given back to the console's TMP. This occurs when the TMP is the highest priority process waiting for the console.

If P_TERM returns to the calling process, the call has failed for one of two reasons. Either the process has the KEEP flag on, or it has the SYSTEM flag on, and the Termination Code is not 0FFH.

P_TERMCPM

Terminate a Calling Process

Entry Parameters:
Register CL: 00H (0)

Returned Values:
Register AX: 0 on success, 0FFFFH on failure
BX: Same as AX
CX: Error Code

P_TERMCPM terminates the calling process, releasing all system resources owned by the process. P_TERMCPM is implemented internally by calling P_TERM with the Termination Code set to 00H.

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

Table 6-5 contains the list of returned error codes.

Queue system calls under Concurrent use the **Queue Parameter Block** (QPB) data structure to pass parameters to and from Concurrent. Figure 6-15 shows the Queue Parameter Block format, and Table 6-21 lists the QPB field elements. Listing 6-3 shows the programming equates for the Queue Parameter Block data structure.

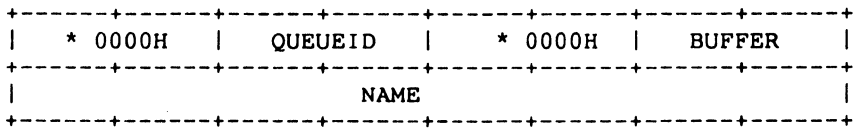


Figure 6-15. QPB - Queue Parameter Block

Table 6-21. QPB Field Definitions

Field	Description
QUEUEID	Queue number field; filled in by Q_OPEN operation.
* 0000H	Reserved for internal use; must be initialized to zero.
BUFFER	Offset address of Queue Message Buffer.
NAME	Name of Queue for Q_OPEN operation.

Listing 6-3. Queue Parameter Block Definition

```
;*****************************************************
;*
;*      QPB - Queue Parameter Block Definition
;*
;*      -
;*  00    0000H   queueid   0000H    buffer
;*      -
;*  08                  name
;*      -
;*
;*      queueid - Queue ID, address of QD
;*      buffer  - address to read/write into/from
;*      name    - name of queue (for open only)
;*
;*****************************************************

qpb_0           equ     word ptr 0
qpb_queueid     equ     word ptr qpb_0 + word
qpb_buffer      equ     word ptr qpb_queueid + 4
qpb_name        equ     byte ptr qpb_buffer + word

qpb_len         equ     qpb_name + qnamsiz
qnamsiz         equ     8
```

Q_CREAD

Conditionally Read A Message From A System Queue

Entry Parameters:
- Register CL: 08AH (138)
- DX: QPB Address - Offset
- DS: QPB Address - Segment
  QPB_queueid filled in by previous Q_OPEN
  QPB_buffer set to message buffer offset

Returned Values:
- Register AX: 0 on success, 0FFFFH on failure
- BX: Same as AX
- CX: Error Code message in buffer

Q_CREAD is analogous to Q_READ, but it returns an error code if there are not enough messages to read, instead of waiting for another process to write to the queue.

Table 6-5 contains the list of error codes returned in CX.

Q_CWRITE

Conditionally Write A Message To A System Queue

Entry Parameters:
- Register CL: 08CH (140)
- DX: QPB Address - Offset
- DS: QPB Address - Segment
  QPB_queueid filled in by previous Q_OPEN
  QPB_buffer set to message buffer offset
  message in current DMA buffer

Returned Values:
- Register AX: 0 on success, 0FFFFH on failure
- BX: Same as AX
- CX: Error Code

Q_CWRITE is analogous to Q_WRITE, but it returns an error code if there is not enough system queue buffer space for the message to be written, instead of waiting for another process to read from the queue.

Table 6-5 contains the list of error codes returned in CX.

Q_DELETE

Delete A System Queue

Entry Parameters:
- Register CL: 088H (136)
- DX: QPB Address - Offset
- DS: QPB Address - Segment
  QPB_queueid filled in by a previous Q_OPEN call

Returned Values:
- Register AX: 0 on success, 0FFFFH on failure
- BX: Same as AX
- CX: Error Code

Q_DELETE removes a system queue from the system, and returns an error code if the queue cannot be deleted or if the queue has not been opened prior to the call.

Table 6-5 contains the list of error codes returned in CX.

Q_MAKE

Make A System Queue

Entry Parameters:
Register CL: 086H (134)
DX: QD Address - Offset
DS: QD Address - Segment
QD filled in

Returned Values:
Register AX: 0 on success, 0FFFFH on failure
BX: Same as AX
CX: Error Code

Q_MAKE creates a system queue for the calling process by passing the address of a **Queue Descriptor** (QD). Figure 6-16 shows the Queue Descriptor format, and Table 6-22 lists the QD field elements.

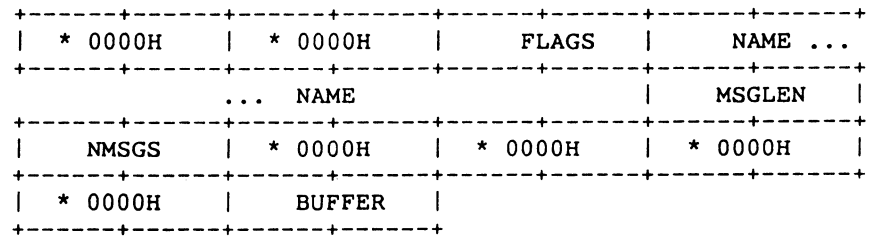


Figure 6-16. QD - Queue Descriptor

Table 6-22. Queue Descriptor Field Definitions

Field	Definition
FLAGS	Queue Flags. The bits are defined as follows: 0001H Mutual exclusion queue 0002H Cannot be deleted 0004H Restricted to system processes 0008H RSP message queue 0010H Used internally 0020H RPL address queue 0040H Used internally 0080H Used internally All remaining flags reserved for future use
NAME	8-byte queue name. All 8 bits of each character are matched on an Q_OPEN call.
MSGLEN	Number of bytes in each logical message.
NMSGS	Maximum number of logical messages to be supported. If the number of messages written to the queue equals this maximum, no more messages are allowed until a message is read.
BUFFER	Address of the queue buffer. This buffer must be (NMSGS * MSGLEN) bytes long. The address is an offset relative to the DS register. This field is unused if the QD resides outside of the System Data Segment. Typically this field is 0 if the queue is being created by a transient program. RSPs that create queues must initialize this field to point to a buffer. The Data Segment of an RSP's queue is considered part of the System Data Segment unless it is beyond 64k of the beginning of the System Data Segment. If BUFFER contains 0FFFFH, Q_MAKE allocates space for the Queue Descriptor and Queue Buffer from the system queue buffer area in the System Data Segment (SYSDAT).
* 0000H	For internal use; must be initialized to zero.

Every system queue is associated with a Queue Descriptor that resides in Concurrent's System Data Segment. If the Queue Descriptor is within the System Data Segment, Concurrent uses it directly for the System Queue. If the Queue Descriptor is outside the System Data Segment, Concurrent obtains a Queue Descriptor from an internal Queue Descriptor table. If there are no unused Queue Descriptors in the internal table, Q_MAKE returns an error code.

Table 6-5 contains the list of error codes returned in CX.

The buffer for a system queue must also reside within the System Data area. For non-00H length buffers, resident buffers are used directly. Concurrent obtains a buffer from the Queue Buffer Area if the buffer does not reside within the System Data Segment. The size of the buffer is calculated from the NMSGS and MSGLEN fields. Q_MAKE returns an error code if there is not enough unused buffer area left to accommodate this new buffer.

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

Under Concurrent, all system queues must be explicitly opened (see Q_OPEN) before being used to read or write messages or to delete the queue.

Q_OPEN

Open A System Queue

Entry Parameters:
- Register CL: 087H (135)
- DX: QPB Address - Offset
- DS: QPB Address - Segment
- QPB_name filled in

Returned Values:
- Register AX: 0 on success, 0FFFFH on failure
- BX: Same as AX
- CX: Error Code
- QPB_queueid filled in

Q_OPEN opens a system queue by examining each existing system queue and attempting to match the name in the QPB with the name of a system queue. All eight bytes of the name must match for a successful open. All bits of each byte are examined.

If the open operation is successful, Q_OPEN modifies the Queue ID Field of the QPB. Once the the queue is opened, subsequent reads, writes, or a delete are allowed.

Note: Under Concurrent, you must use Q_OPEN to explicitly open all system queues before attempting a read, write, or delete operation.

Table 6-5 contains the list of error codes returned in CX.

Q_READ

Read A Message From A System Queue

Entry Parameters:
- Register CL: 089H (137)
- DX: QPB Address - Offset
- DS: QPB Address - Segment
  QPB_queueid filled in by previous Q_OPEN
  QPB_buffer set to message buffer offset

Returned Values:
- Register AX: 0 on success, 0FFFFH on failure
- BX: Same as AX
- CX: Error Code message in buffer

Q_READ reads a message from a system queue that was previously opened by the calling process. Q_READ returns an error code if the queue was not previously opened or if the system queue has been deleted since the Q_OPEN call. If there are not enough messages to read from the queue, the calling process waits until another process writes into the queue before returning.

Table 6-5 contains the list of error codes returned in CX.

Q_WRITE

Write A Message To A System Queue

Entry Parameters:
- Register CL: 08BH (139)
- DX: QPB Address - Offset
- DS: QPB Address - Segment
  QPB_queueid filled in by previous Q_OPEN
  QPB_buffer set to message buffer offset message in buffer

Returned Values:
- Register AX: 0 on success, 0FFFFH on failure
- BX: Same as AX
- CX: Error Code

Q_WRITE writes a message to a system queue that was previously opened by the calling process. Q_WRITE returns an error code if the queue was not previously opened or if the system queue has been deleted since the Q_OPEN call. If there is not enough buffer space in the queue, the calling process waits until another process reads from the queue before writing to the queue and returning.

Table 6-5 contains the list of error codes returned in CX.

S_BDOSVER

Return BDOS Version Number

Entry Parameters:
Register CL: 0CH (12)

Returned Values:
Register AX: BDOS Version Number
BX: Same as AX

S_BDOSVER returns the BDOS file system version number, allowing version-independent programming. Register AX values are:

01432H - Version 3.2
01441H - Version 4.1
01450H - Version 5.0

S_BIOS

Call BIOS Character Routine

Entry Parameters:
Register CL: 032H (50)
DX: BIOS Descriptor Address - Offset
DS: BIOS Descriptor Address - Segment

Returned Values:
Register AX: BIOS Return
BX: Same as AX

S_BIOS provides compatibility for programs generated under CP/M-86 that use this system call (Function 50). The calling process passes the address of a **BIOS Descriptor**. Figure 6-17 shows the format.

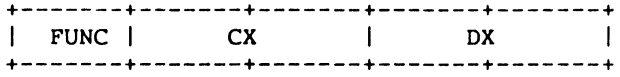


Figure 6-17. BIOS Descriptor Format

Since Concurrent only supports routines that interface with character devices, the arguments to character routines such as CONIN and LIST must be converted to those appropriate for Concurrent's XIOS. Refer to the System Guide for further information about the XIOS.

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

S_OSVER

Return the Current Version Of Concurrent DOS 86 System

Entry Parameters:
Register CL: 0A3H (163)

Returned Values:
AX: Version Number
BX: Same as AX
CX: Error Code

S_OSVER returns the version number of the operating system, allowing version-independent programming. Register AX values are:

01432H - Version 3.2
01441H - Version 4.1
01450H - Version 5.0

Table 6-5 contains the list of error codes returned in CX.

S_SERIAL

Return the Serial Number

Entry Parameters:
Register CL: 06BH (107)
DX: SERIAL Address - Offset
DS: SERIAL Address - Segment

Returned Values:
SERIAL filled in

S_SERIAL returns the Concurrent DOS 86 serial number to the addressed, six-byte SERIAL field as a six-byte ASCII numeral. Figure 6-18 shows the format of the returned value.

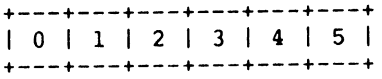


Figure 6-18. SERIAL Number Format

S_SYSDAT

Return Address Of The System Data Segment

Entry Parameters:
Register CL: 09AH (154)

Returned Values:
AX: Sysdat Address - Offset
BX: Same as AX
ES: Sysdat Address - Segment

S_SYSDAT returns the address of the **System Data Segment** of the calling process. The System Data Segment contains all Process Descriptors, Queue Descriptors, the roots of system lists, and other internal data.

Figure 6-19 shows the SYSDAT Table, and Table 6-23 lists its fields.

00H	SUP ENTRY					RESERVED		
08H	RESERVED							
28H	XIOS ENTRY					XIOS INIT		
30H	RESERVED							
38H	DISPATCHER					PDISP		
40H	CCPMSEG		RSPSEG		ENDSEG		RSVD	NVCNS
48H	NLCB	NCCB	N_ FLAGS	SYS_ DISK	MMP		RSVD	DAY FILE
50H	TEMP DISK	TICKS /SEC	LUL		CCB		FLAGS	
58H	MDUL		MFL		PUL		QUL	
60H	QMAU							
68H	RLR		DLR		DRL		PLR	
70H	RESERVED		THRDRT		QLR		MAL	
78H	VERSION		VERNUM		CCPMVERNUM		TOD_DAY	
80H	TOD _HR	TOD _MIN	TOD _SEC	NCON DEV	NLST DEV	NCIO DEV	LCB	
88H	OPEN_FILE		LOCK_ MAX	OPEN_ MAX	OWNER_8087		ACB	
90H	RESERVED							
98H	MWDR		RESERVED		NACB	PSD	RSVD	XPCNS
A0H	OFF_8087		SEG_8087		RESERVED			

Figure 6-19. SYSDAT Table

Table 6-23. SYSDAT Table Data Fields

Field	Definition
SUP ENTRY	Double-word address of the Supervisor entry point for intermodule communication. All internal system calls go through the SUP entry point.
XIOS ENTRY	Double-word address of the XIOS entry point for intermodule communication. All XIOS function calls go through the XIOS entry point.
XIOS INIT	Double-word address of the XIOS Initialization entry point. System hardware initialization takes place by a call through this entry point.
DISPATCHER	Double-word address of the Dispatcher entry point that handles interrupt returns. Executing a JMPF instruction to this address is equivalent to executing an IRET instruction. The Dispatcher routine causes a dispatch to occur and then executes an IRET. All registers are preserved and one level of stack is used. This location should be used as an exit point by all XIOS interrupt handlers that use the DEV_SETFLAG call.
PDISP	Double-word address of the Dispatcher entry point that causes a dispatch to occur with all registers preserved. Once the dispatch is done, a RETF instruction is executed. Executing a JMPF PDISP is equivalent to executing a RETF instruction. This location should be used as an exit point whenever the XIOS releases a resource that a waiting process might want.
CCPMSEG	Starting paragraph of the operating system area. This is also the Code Segment of the Supervisor Module.
RSPSEG	Paragraph Address of the first RSP in a linked list of RSP Data Segments. The first word of the data segment points to the next RSP in the list. Once Concurrent has been initialized, this field is zero.
ENDSEG	First paragraph beyond the end of the operating system area, including any buffers consisting of uninitialized RAM allocated to the operating system by GENCCPM. These include the Directory Hashing, Disk Data, and XIOS ALLOC buffers. These buffer areas, however, are not part of the CCPM.SYS file.

Table 6-23. (Cont'd)

Field	Definition
NVCNS	Number of virtual consoles, copied from the XIOS Header by GENCCPM.
NLCB	Number of List Control Blocks, copied from the XIOS Header by GENCCPM.
NCCB	Number of Character Control Blocks, copied from the XIOS Header by GENCCPM.
NFLAGS	Number of system flags as specified during GENCCPM.
SYSDISK	Default system disk. P_CLI looks on this disk if it cannot open the command file on the user's current default disk. Set during GENCCPM. Concurrent initializes offset 4BH to logical drive P, which is the permanent system disk identifier that can point to different physical drives.
MMP	Maximum memory allowed per process. Set during GENCCPM.
DAY FILE	Day File option. If this field is 0FFH, Concurrent displays file logging information on system consoles at each command. Set during GENCCPM.
TEMP DISK	Default temporary disk. Programs that create temporary files should use this disk. Set during GENCCPM.
TICKS/SEC	The number of system ticks per second.
LUL	Locked Unused List. Linked list root of unused Lock list items.
CCB	Address of the Character Control Block Table, copied from the XIOS Header by GENCCPM.
FLAGS	Address of the Flag Table.
MDUL	Memory Descriptor Unused List. Linked list root of unused Memory Descriptors.
MFL	Memory Free List. Linked list root of free memory partitions.
PUL	Process Unused List. Linked list root of unused Process Descriptors.
QUL	Queue Unused List. Linked list root of unused Queue Descriptors.
QMAU	Queue Buffer Memory Allocation Unit.

Table 6-23. (Cont'd)

Field	Definition
RLR	Ready List Root. Linked list of PDs that are ready to run.
DLR	Delay List Root. Linked list of PDs that are delaying for a specified number of system ticks.
DRL	Dispatcher Ready List. Temporary holding place for PDs that have just been made ready to run.
PLR	Poll List Root. Linked list of PDs that are polling on devices.
THRDRT	Thread List Root. Linked list of all current PDs on the system. The list is threaded though the THREAD field of the PD instead of the LINK field.
QLR	Queue List Root. Linked list of all System QDs.
MAL	Memory Allocation List. Linked list of active memory allocation units. A MAU is created from one or more memory partitions.
VERSION	Address, relative to CCPMSEG, of ASCII version string.
VERNUM	Concurrent file system version number (returned by S_BDOSVER).
CCPMVERNUM	Concurrent version number (returned by S_OSVER).
TOD_DAY	Time-of-Day. Number of days since 1 Jan, 1978.
TOD_HR	Time-of-Day. Hour of the day.
TOD_MIN	Time-of-Day. Minute of the hour.
TOD_SEC	Time-of-Day. Second of the minute.
NCONDEV	Number of XIOS consoles, copied from the XIOS Header by GENCCPM.
NLSTDEV	Number of XIOS list devices, copied from the XIOS Header by GENCCPM.
NCIODEV	Total number of character devices (NCONDEV + NLSTDEV).
LCB	Offset of the List Control Block Table, copied from the XIOS Header by GENCCPM.
OPEN_FILE	Open File Drive Vector. Designates drives with open files. Each set bit of the word value represents a disk drive containing open files; the least significant bit represents Drive A, the most significant, Drive P.

Table 6-23. (Cont'd)

Field	Definition
LOCK_MAX	Maximum number of locked records per process. Set during GENCCPM.
OPEN_MAX	Maximum number of open disk files per process. Set during GENCCPM.
OWNER_8087	Specifies 8087 information. If set to 0FFFFH, Concurrent assumes there is no 8087 in the system. If set to 0, there is an 8087 but no process owns it. If set to any other value, Concurrent assumes the value is the PD offset of the 8087 current process.
ACB	Address of Auxiliary Control Block Table copied from XIOS Header by GENCCPM.
MWDR	Memory Window Desriptor Root. Linked list root of Memory Window Descriptors that describe the location of available logical address windows for mapping to physical memory pages.
NACB	Number of Auxiliary Control Blocks. Copied by GENCCPM from XIOS Header.
PSD	Physical search disk. The default system disk as copied from SYSDISK field by Supervisor (SUP) initialization routine. The BDOS uses PSD as the initial drive for drives N and O and the system drive, P. Concurrent searches the system drive whenever it cannot find a file on the default drive.
XPCNS	Specifies the number of physical consoles.
OFF_8087	Offset of the hardware-dependent 8087 interrupt vector in low memory. If you supply your own 8087 exception handler routine, store its offset at this address.
SEG_8087	Segment address of the hardware-dependent 8087 interrupt vector in low memory. If you supply your own 8087 exception handler routine, store its segment at this address.

T_GET

Get System Time And Date

Entry Parameters:
Register CL: 69H (105)
DX: TOD Address - Offset
DS: TOD Address - Segment

Returned Values:
AL: Seconds
TOD filled in
(Days, Hours and Minutes only)

T_GET obtains the system internal time and date. The calling process passes the address of a four-byte **Time of Day** (TOD) data structure that receives the time and date values. Figure 6-20 shows the Time of Day structure, and Table 6-24 lists the TOD field definitions.

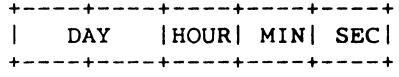


Figure 6-20. TOD - Time-of-Day Structure

Table 6-24. Time-of-Day Field Definitions

Field	Definition
DAY	The number of days since 1 January 1978. The day is stored as a 16-bit integer.
HOUR	The current hour of the current day. The hour is represented as a 24 hour clock in 2 binary coded decimal (BCD) digits.
MIN	The current minute of the current hour. The minute is stored as 2 BCD digits.
SEC	The current second of the current minute. The second is stored as 2 BCD digits.

T_GET is equivalent to T_SECONDS, except that it does not return the SECONDS field of the internal time.

T_SECONDS

Get Current System Time And Day

Entry Parameters:
Register CL: 09BH (155)
DX: TOD Address - Offset
DS: TOD Address - Segment

Returned Values:
TOD filled in
(Days, Hours, Minutes, and Seconds)

T_SECONDS returns the current encoded time and date (including seconds) in the TOD structure passed by the calling process. See T_GET for the format of the TOD structure.

T_SET

Set System Time And Date

Entry Parameters:
Register CL: 068H (104)
DX: TOD Address - Offset
DS: TOD Address - Segment

T_SET sets the system internal time and date. The calling process passes the address of a 4-byte TOD structure containing the time and date specification. See T_GET for the format of the TOD structure.

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

T_SET also sets the second field of the system time and date to 00H.

End of Section 6

PC DOS System Calls

This section describes the interface that allows transient programs to emulate DOS system functions under Concurrent. DOS system call error return codes are also described in this section.

7.1 Introduction

The DOS system calls supported by Concurrent are divided into the following categories:

* Character Device I/O
* File Management
* Extended File Management
* Directory Management
* Miscellaneous
* Program Control
* Memory Management
* Time

Table 7-1 summarizes the DOS system calls according to these categories.

Note: You should not code any program using a mix of DOS calls with the native Concurrent system calls as described in Section 6; code only for a pure DOS or a pure Concurrent environment. CMD files are allowed to make call only native CDOS calls; EXE or (COM) files are allowed to make only DOS calls.

Table 7-1. DOS System Call Categories

Hex Number	System Call
	Character Device I/O
01	Keyboard Input
02	Console Output
03	Auxiliary Input
04	Auxiliary Output
05	Printer Output
06	Direct Console I/O
07	Direct Console Input
08	Console Input without Echo
09	Print String
0A	Buffered Keyboard Input
0B	Check Console Status
0C	Character Input with Buffer Flush
33	Ctrl-Break Check
	File Management
0D	Disk Reset
0E	Select Disk
0F	Open File
10	Close File
11	Search for First Entry
12	Search for Next Entry
13	Delete File
14	Sequential Read
15	Sequential Write
16	Create File
17	Rename File
19	Current Disk
1A	Set Disk Transfer Address
1B	Allocation Table Address
1C	Allocation Table for Specific Drive
21	Random Read
22	Random Write
23	File Size
24	Set Random Record Field
27	Random Block Read
28	Random Block Write
29	Parse File Name

Table 7-1. (Cont'd)

Hex Number	System Call
2E	Set/Reset Verify Switch
2F	Get Disk Transfer Address
36	Get Disk Free Space
54	Get Verify State
	Extended File Management
3C	Create a File (CREAT)
3D	Open a File Handle
3E	Close a File Handle
3F	Read from a File or Device
40	Write to a File or Device
41	Erase a File from Directory (UNLINK)
42	Move File Read/Write Pointer (LSEEK)
43	Change File Mode (CHMOD)
45	Duplicate a File Handle (DUP)
46	Force a Duplicate of a File Handle (DUP)
4E	Find First
4F	Find Next
56	Rename a File
57	Get/Set File Time and Date Stamps
	Directory Management
39	Create a Subdirectory (MKDIR)
3A	Remove a Subdirectory (RMDIR)
3B	Change Current Directory (CHDIR)
47	Get Current Directory
	Miscellaneous
25	Set Vector
30	Get DOS Version Number
35	Get Vector
38	Get Country Dependent Information

Table 7-1. (Cont'd)

Hex Number	System Call
	Program Control
00	Program Terminate
26	Create a New Program Segment
31	Keep Process
4B	Execute a Program (EXEC)
4C	Terminate a Process (EXIT)
4D	Get Subprocess Return Code (WAIT)
	Memory Management
48	Allocate Memory
49	Free Allocated Memory
4A	Modify Allocated Memory Blocks (SETBLOCK)
	Time
2A	Get Date
2B	Set Date
2C	Get Time
2D	Set Time

7.2 DOS System Call Parameters

Under Concurrent, a process requests a DOS function by placing the function number in register AH, supplying additional information in other registers as necessary, and then issuing an INT 21H. When Concurrent takes control, it switches to an internal stack. User registers, except AX, are preserved unless information is passed back to the register as indicated in the specific requests. To accommodate the interrupt system, the user stack should be 80H in addition to the program's needs.

7.2.1 ASCIIZ Input Strings

The DOS system calls listed in Table 7-2 require the address of an **ASCIIZ** string in registers DS:DX as an input parameter. ASCIIZ strings are ASCII character strings containing an optional drive reference and directory path. In some cases, the strings also include a file name. ASCIIZ strings are delimited by a zero byte or a null (0H). Path names can be delimited by either a slash (/) or a backslash (\) character.

Table 7-2. DOS System Calls Requiring ASCIIZ Strings

Hex Number	DOS System Call
39	Create a Subdirectory (MKDIR)
3A	Remove a Subdirectory (RMDIR)
3B	Change Current Directory (CHDIR)
3C	Create a File
3D	Open a File
41	Erase a File from Directory (UNLINK)
43	Change File Mode (CHMOD)
4B	Execute a Program (EXEC)
4E	Find First
56	Rename a File

7.2.2 DOS File and Device Handles

Create a File (3CH), Open a File (3DH), and Duplicate a File Handle (45H) return a 16-bit binary identifier value in register AX. This identifier is called a "handle." The handle is used to later identify files or devices that have been opened or created with the system call that originally returned it.

Table 7-3 lists the handles that are predefined by DOS to identify standard devices. Your program can use these handles without previously opening the devices assigned to them.

Table 7-3. DOS Standard Device Handles

Handle	Standard Device
0000	Input device
0001	Output device
0002	Error output device
0003	Auxiliary device
0004	List device

7.3 DOS System Call Error Return Codes

When a DOS call operation is successful, most calls clear the carry flag. If there is an error, these calls set the carry flag and return an error code in the AX register. Table 7-4 lists the binary error codes returned in register AX.

Table 7-4. DOS System Call AX Error Codes

Code	Error
0	No error
1	Illegal function
2	File not found
3	Path not found
4	No file handles (too many open files)
5	Access denied
6	Illegal file handle
7	Memory control blocks destroyed
8	Insufficient memory
9	Illegal memory block address
10	Invalid environment
11	Illegal format
12	Invalid access code
13	Invalid data
15	Invalid drive specified
16	Removal of current directory attempted
17	Devices do not match
18	No more files

7.4 DOS System Call Summary

Table 7-5 lists the DOS system functions supported by Concurrent. The table includes the parameters a process must pass to the call and the values the call returns to the process. Table 7-5 also lists the page references where each call is described in detail. At the end of Table 7-5 is a list of conventions used in the system call summary.

Table 7-5. DOS System Call Summary

Hex	Function	Input Parameters	Returned Values	Page
00	Program Terminate	CS = .PSP	none	7-83
01	Keyboard Input	none	AL = char/00	7-13
02	Console Output	DL = char	none	7-14
03	Auxiliary Input	none	AL = char	7-15
04	Auxiliary Output	DL = char	none	7-16
05	Printer Output	DL = char	none	7-17
06	Direct Console I/O	DL = FF/char	AL = char/00	7-18
07	Direct Console Input	none	AL = char/00	7-19
08	Console Input w/o Echo	none	AL = char/00	7-20
09	Print String	DS:DX = .string	none	7-21
0A	Buffered Console Input	DS:DX = .Buffer	none	7-22
0B	Check Console Status	none	AL = FF/00	7-23
0C	Character Input with Buffer Flush	AL = Call #	none	7-24
0D	Disk Reset	none	AX = Err Code	7-31
0E	Select Disk	DL = Drive Vect	AL = # of Drives	7-32
0F	Open File	DS:DX = .FCB	AL = 00/FF	7-33
10	Close File	DS:DX = .FCB	AL = 00/FF	7-34
11	Search for First Entry	DS:DX = .FCB	AL = 00/FF	7-35
12	Search for Next Entry	DS:DX = .FCB	AL = 00/FF	7-37

Table 7-5. (Cont'd)

Hex	Function	Input Parameters	Returned Values	Page
13	Delete File	DS:DX = .FCB	AL = 00/FF	7-38
14	Sequential Read	DS:DX = .FCB	AL = 00/01/02/03	7-39
15	Sequential Write	DS:DX = .FCB	AL = 00/01/02	7-40
16	Create File	DS:DX = .FCB	AL = 00/FF	7-41
17	Rename File	DS:DX = .FCB	AL = 00/FF	7-42
19	Current Disk	none	AL = Cur Drive #	7-43
1A	Set DTA	DS:DX = .DTA	none	7-44
1B	Allocation Table Address	none	DS:BX = .FAT ID DX = # of Clusters AL = Sects/Cluster CX = Size of Sect	7-45
1C	Allocation Table Address for a Specific Drive	DL = Drive #	DS:BX = .FAT ID DX = Alloc Units DL = Sects/Unit CX = Size of Sect	7-46
21	Random Read	DS:DX = .FCB	AL = 00/01/02/03	7-47
22	Random Write	DS:DX = .FCB	AL = 00/01/02	7-48
23	File Size	DS:DX = .FCB	AL = 00/FF	7-49
24	Set Random Record Field	DS:DX = .FCB	none	7-50
25	Set Vector	DS:DX = .Intrpt Routine AL = Interrupt	none	7-78
26	Create New Program Segment	DX = Seg Num	none	7-84
27	Random Block Read	DS:DX = .FCB CX = Rec Count	AL = 00/01/02/03 CX = # of Recs	7-51

Table 7-5. (Cont'd)

Hex	Function	Input Parameters	Returned Values	Page
28	Random Block Write	DS:DX = .FCB CX = Rec Count/00	AL = 00/01/02 CX = # Recs	7-52
29	Parse File Name	DS:SI = .Com Line ES:DI = .FCB AL = Bit Map	AL = 00/01/FF DS:SI = 1st Char ES:DI = .FCB	7-53
2A	Get Date	none	AL = Day (0-6) CX = Year (Binary) DX = Month/Day	7-96
2B	Set Date	CX = Year DX = Month/Day	AL = 00/FF	7-97
2C	Get Time	none	CX = Hours/Mins DX = Secs/100's	7-98
2D	Set Time	CX = Hours/Mins DX = Secs/100's	AL = 00/FF	7-99
2E	Set/Reset Verify Switch	DL = 0 AL = 00/01	none	7-55
2F	Get DTA	none	ES:BX = .DTA	7-56
30	Get DOS Version #	none	AX = Vers # BX = 0000 CX = 0000	7-79
31	Keep Process	AL = Exit Code DX = Paras	none	7-85
33	Ctrl-Break Check	AL = 00/01 DL = 00/01	DL = 00/01	7-25
35	Get Vector	AL = Interrupt	ES:BX = .Vect	7-80

Table 7-5. (Cont'd)

Hex	Function	Input Parameters	Returned Values	Page
36	Get Disk Free Space	DL = Drive #	BX = # of Clusts DX = Total Clusts CX = Bytes/Sect AX = Sects/Clust	7-57
38	Get Country Dependent Information	DS:DX = .Block AL = 00	see def	7-81
39	Create Subdirectory	DS:DX = .ASCIIZ	AX = Err Code	7-74
3A	Remove Subdirectory	DS:DX = .ASCIIZ	AX = Err Code	7-75
3B	Change Current Directory	DS:DX = .ASCIIZ	AX = Err Code	7-76
3C	Create a File	DS:DX = .ASCIIZ CX = Attrib	AX = Handle	7-59
3D	Open a File Handle	DS:DX = .ASCIIZ AL = Access Code	AX = Handle	7-60
3E	Close a File Handle	BX = Handle	AX = Err Code	7-61
3F	Read from a File or Device	BX = Handle CX = # of bytes DS:DX = .Buffer	AX = # of bytes read	7-62
40	Write to a File or Device	BX = Handle CX = # of bytes DS:DX = .Buffer	AX = # of bytes written	7-63
41	Delete a File from a Directory	DS:DX = .ASCIIZ	AX = Err Code	7-64
42	Move File R/W Pointer (LSEEK)	AL = Method Val BX = Handle CX:DX = Offset	DX:AX = Pointer	7-65
43	Change File Mode (CHMOD)	AL = Funct Code DS:DX = .ASCIIZ	CX = Attribute	7-66

Table 7-5. (Cont'd)

Hex	Function	Input Parameters	Returned Values	Page
45	Duplicate a File Handle	BX = Handle	AX = Dup Handle	7-67
46	Force a Handle Duplicate	BX = Handle CX = 2nd Handle	AX = Err Code	7-68
47	Get Current Directory	DL = Drive # DS:SI = .Path	AX = Err Code	7-77
48	Allocate Memory	BX = # of Paras	AX = .Block BX = Max Size	7-93
49	Free Allocated Memory	ES = Block Seg	AX = Err Code	7-94
4A	Modify Memory Blocks (SETBLOCK)	ES = Block Seg BX = Block size	AX = Err Code BX = Max Size	7-95
4B	Load or Execute a Program (EXEC)	DS:DX = .ASCIIZ ES:BX = .LPB AL = Funct Value	AX = Err Code	7-86
4C	Terminate a Process (EXIT)	AL = Return Code	none	7-91
4D	Get Subprocess Return Code	none (WAIT)	AX = Exit Code	7-92
4E	Find First	DS:DX = .ASCIIZ CX = File Attribute	AX = Err Code	7-69
4F	Find Next	none	AX = Err Code	7-70
54	Get Verify State	none	AL = 00/01	7-58
56	Rename a File	DS:DX = .ASCIIZ ES:DI = .ASCIIZ (New Name)	AX = Err Code	7-71

Table 7-5. (Cont'd)

Hex	Function	Input Parameters	Returned Values	Page
57	Get/Set File Time and Date Stamps	AL = 00 (Get) = 01 (Set) BX = Handle CX = Time (if Set) DX = Date (if Set)	CX = Time Stamp DX = Date Stamp	7-72

Conventions used in Table 7-5:

.	Address of
#	Number
Alloc	Allocation
ASCIIZ	ASCIIZ String
char	ASCII Character
charstrng	ASCII Character String
Clus	Cluster
Com	Command
Cur	Current
Dir	Directory
DTA	Disk Transfer Area
Err	Error
FAT	File Allocation Table
FCB	File Control Block
FN	File Name
Handle	File or Device Handle
Intrpt	Interrupt
LPB	Load Parameter Block
Para	Paragraph
PSP	Program Segment Prefix
Rec	Record
Sect	Sector
Seg	Segment
Vect	Vector

Keyboard Input (01H)

Read Characters from Default Input Device

Entry Parameters:
Register AH: 01H

Returned Values:
Register AL: ASCII character or 00H
(First call to read extended ASCII code.)

Keyboard Input reads a character from the default input device of the calling process, writes the character to the default console device, and then returns the character in the AL register. If a character is not ready to be read, Keyboard Input waits for one before returning to the calling process.

Keyboard Input executes an INT 23H when it reads Ctrl-Break. When the calling process wants to read an extended ASCII code character, it must call Keyboard Input twice. Keyboard Input returns 00H in register AL to indicate that a subsequent call will return an extended ASCII code character.

Console Output (02H)

Write Characters to Default Console Device

Entry Parameters:
Register AH: 02H
DL: ASCII character

Console Output writes the character in DL to the calling process's default console device. If DL contains a backspace character (08H), Console Output moves the cursor left one position, writes a space at that location, and leaves the cursor there.

As with the Keyboard Input system call, Console Output executes an INT 23H when the user has entered Ctrl-Break.

Auxiliary Input (03H)

Reads a Character from Default Auxiliary Input Device

Entry Parameters:
Register AH: 03H

Returned Values:
Register AL: ASCII character

Auxiliary Input reads the next ASCII character from the default auxiliary device and returns it in register AL.

Auxiliary Output (04H)

Write a Character to Default Auxiliary Output Device

Entry Parameters:
Register AH: 04H
DL: ASCII character

Auxiliary Output writes the ASCII character specified in register DL to the default auxiliary device.

Printer Output (05H)

Write a Character to the Default Printer Device

Entry Parameters:
Register AH: 05H
DL: ASCII character

Printer Output writes the ASCII character specified in register DL to the default printer device.

Direct Console I/O (06H)

Perform Direct Console I/O to Default Input Device

Entry Parameters:
Register AH: 06H
DL: FFH (Input)
ASCII character (Output)

Returned Values:
Register AL: ASCII character or 00H
(First call to read extended ASCII code.)

When register DL contains FFH, the Direct Console I/O call reads a character from the calling process's default input device, clears the zero flag, and returns the character in register AL. If a character is not ready from the input device, Direct Console I/O sets the zero flag and returns 00H in AL.

If register DL contains anything other than FFH, Direct Console I/O assumes that DL contains a valid character to be written to the default output device. Direct Console I/O does not check for Ctrl-PrtSc or Ctrl-Break input from the console.

When the calling process wants to read or write an extended ASCII code character, it must call Direct Console I/O twice. The first call returns 00H in register AL to indicate that a subsequent call will return an extended ASCII code character.

Note: In a Concurrent environment, you should not use DL = 0FFH if you want to wait for a character; use Function 07H instead.

Direct Console Input (07H)

Perform Direct Console Input to Default Input Device

Entry Parameters:
Register AH: 07H

Returned Values:
Register AL: ASCII character or 00H
(First call to read extended ASCII code.)

The Direct Console Input call reads a character from the default input device of the calling process and returns the character in register AL. If a character is not ready to be read from the input device, the call waits for one before returning to the calling process.

When the calling process wants to read an extended ASCII code character, it must make this call twice. Direct Console Input returns 00H in register AL to indicate that a subsequent call will return an extended ASCII code character.

Direct Console Input does not check for Ctrl-PrtSc or Ctrl-Break input from the console.

Console Input Without Echo (08H)

Read Character From Default Input Device

Entry Parameters:
Register AH: 08H

Returned Values:
Register AL: ASCII character or 00H
(First call to read extended ASCII code.)

The Console Input without Echo call reads a character from the default input device of the calling process and returns the character in register AL. If a character is not ready to be read, the call waits for one before returning to the calling process.

Console Input without Echo executes an INT 23H when it reads Ctrl-Break.

When the calling process wants to read an extended ASCII code character, it must make this call twice. In this case, the call returns 00H in register AL to indicate that the next call will return an extended ASCII code character.

Print String (09H)

Send a Character String to Default Console Device

Entry Parameters:
Register AH: 09H
DS: String address - Segment
DX: String address - Offset

Print String sends each character in the ASCII string addressed by DS:DX and terminated by a $ character (24H) to the calling process's default console device. If the string contains a backspace character (08H), Print String shifts the cursor left one position and writes a space (destructive backspace).

As with the Display Output (02H) call, Print String executes an INT 23H when the user enters Ctrl-Break after the string is written to the console.

Buffered Console Input (0AH)

Read Characters from Default Input Device into Input Buffer

Entry Parameters:
Register AH: 0AH
DS: Buffer address - Segment
DX: Buffer address - Offset

Buffered Console Input reads characters from the calling process's default input device and writes them to the input buffer addressed by registers DS:DX. Figure 7-1 shows the format of the input buffer.

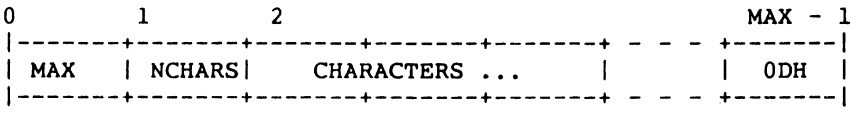


Figure 7-1. DOS Console Buffer Format

The first byte (MAX) specifies the number of characters the buffer can hold. Concurrent sets the second byte (NCHARS) to the number of characters received. This value does not include the terminator character, a carriage return (0DH). Buffered Console Input writes the characters it reads from the input device into the buffer beginning at the third byte. When the call reads a carriage return, its stops writing to the buffer.

When the input buffer contains MAX-1 characters, Buffered Console Input ignores additional input and sounds the bell until it reads a carriage return.

Check Console Status (0BH)

Obtain Status of Default Input Device

Entry Parameters:
Register AH: 0BH

Returned Values:
Register AL: FFH or 00H (No character available)

Check Console Status returns FFH in register AL when a character is available from the calling process's default input device. If the call returns 00H in AL, a character was not ready. Check Console Status executes an INT 23H when it detects a Ctrl-Break character.

Note: In a Concurrent environment, you should not make a status call in a loop; use Function 01H instead.

Character Input with Buffer Flush (0CH)

Flush Type-ahead Buffer and Read
Character(s) From Default Input Device

Entry Parameters:
Register AH: 0CH
AL: Input function number

Returned Values:
Register AL: 00H (No input function performed)

The Character Input with Buffer Flush flushes the type-ahead buffer of the default input device and invokes the input call whose number is entered in register AL.

The calling process can pass the following system call numbers:

- 01H - Keyboard Input
- 06H - Direct Console I/O
- 07H - Direct Console Input
- 08H - Console Input without Echo
- 0AH - Buffered Console Input (see Function 0AH for other parameters)

Ctrl-Break Check (33H)

Check Status or Set Ctrl-Break Checking Mechanism

Entry Parameters:
Register AH: 33H
AL: 00H - Get Ctrl-Break State or
01H - Set Ctrl-Break State
DL: 00H - Set Ctrl-Break Off
01H - Set Ctrl-Break On

Returned Values:
Register DL: 00H - Ctrl-Break Off
01H - Ctrl-Break On

The Ctrl-Break Check call can get or set the current state of Ctrl-Break checking. If register AL = 00H, Ctrl-Break Check returns the current state of the Ctrl-Break checking mechanism in register DL.

If AL contains 01H on entry, Ctrl-Break checking is set according to the value passed in register DL.

7.5 DOS FCB Oriented File Management

DOS programs can use either a standard or an extended FCB. Section 7.5.1 describes the standard DOS FCB. Section 7.5.2 describes the extended FCB. The attribute byte is defined in Section 7.5.3. Section 7.5.4 describes the DOS Disk Transfer Area (DTA).

7.5.1 Standard DOS FCB

Figure 7-2 illustrates the standard DOS FCB. Table 7-6 defines the fields in the standard FCB. The fields from offset 10H through offset 2FH (FILESIZE, DATE, and RESERVED) are set by the Concurrent and must not be changed by user programs. All remaining fields of the standard DOS FCB must be set by user programs.

All data in word fields are stored with the least significant byte first.

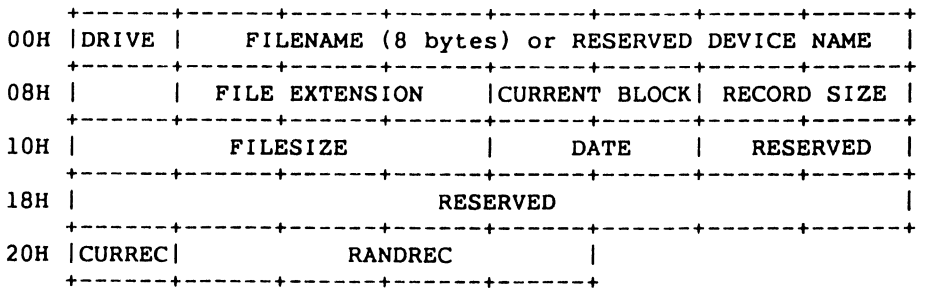


Figure 7-2. DOS Standard File Control Block

Table 7-6. DOS Standard FCB Fields

Field	Definition
DRIVE	Drives are numbered sequentially starting with zero. Before a file is opened, 0 refers to the default drive, 1 to drive A, 2 to drive B, and so forth. After a file is opened, zero is replaced by the actual drive number.
FILENAME	This field may contain either a filename or a reserved name for a device. Filenames must be left-justified with trailing blanks. If a device name is placed in this field, do not include the optional colon.
FILE EXTENSION	Must be left-justified with trailing blanks or all blanks.
CURRENT BLOCK	Current block number relative to the beginning of the file, starting with zero. The Open call sets this field to zero. A block is 128 records long. Record size is specified in the following field. The current block number is used with the current record field (see CURREC, below) for sequential reads and writes.
RECORD SIZE	Logical record size in bytes. The Open call sets this field to 80H. If your records are not 80H bytes, you must set this value to the correct value for your file. Concurrent uses this field to determine locations in the file for all disk reads and writes.
FILE SIZE	File size in bytes. The first word of this two-byte field is the low-order part of the size.
DATE	Date the file was created or last updated. The month, day, and year are mapped in the bits of this field as follows: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 y y y y y y y m m m m d d d d d where: mm is 1-12 dd is 1-31 yy is 0-119 (1980-2099)
RESERVED	Reserved for use by Concurrent.

Table 7-6. (Cont'd)

Field	Definition
CURREC	Current record number in the range 0-127 within the current block (see CURRENT BLOCK, above). You must set this field before sequential read or write operations. To read the first record of a file, set CURREC to zero. This field is not initialized by the Open File (0FH) call.
RANDREC	Record number relative to the beginning of the file, starting with zero. You must set this field before random read/write operations. This two-word field contains the low-order part of the record number in the first word. If a file's record size is less than 64 bytes, both words are used. Otherwise, only the first three bytes are used. This field is not initialized by the Open File call. If you use the FCB at offset 5CH in the Program Segment Prefix (PSP), the last byte of the RANDREC field overlaps the first byte of the unformatted parameter area in the PSP.

An unopened FCB consists of the FCB prefix (for extended FCBs), the drive number, and the filename and extension. An opened FCB is one in which the remaining fields have been filled in by Create File or Open File calls.

7.5.2 DOS Extended FCB

Programs can use extended FCBs to create or search for files that have special attributes.

An extended FCB adds a seven-byte prefix to a normal FCB. Figure 7-4 shows the format of the FCB prefix. Numbers are offsets from the beginning of a normal FCB.

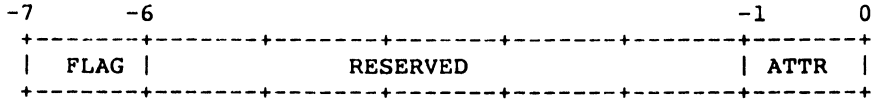


Figure 7-3. DOS Extended FCB Prefix

The fields in the extended FCB are defined in Table 7-7.

Table 7-7. DOS Extended FCB Fields

Field	Definition
FLAG	Byte containing FFH to indicate an extended FCB.
RESERVED	Reserved for use by Concurrent.
ATTR	Attribute byte. The attribute byte is defined in Section 7.5.3.

7.5.3 DOS File Attribute Byte

A program can assign one or more attributes to a file at the time the file is created. The **attribute byte**, as it appears in a directory entry or in the prefix of an extended DOS FCB, can have the values listed in Table 7-8.

Table 7-8. DOS Attribute Byte Values

Value	Meaning
00H	Indicates a normal file. These files have no special attributes and are not excluded from any directory search.
01H	Indicates that the file is marked read/only. If you use the Open a File Handle call (3DH) to attempt to open a read/only file, the call returns an error.
02H	Indicates a hidden file. A file with this attribute is excluded from normal directory searches.
04H	Indicates a system file. A file with this attribute is excluded from normal directory searches.
08H	Indicates that an entry contains a volume label in the filename and extension fields. When the attribute byte has this value, a directory entry contains no other usable information and can exist only in the root directory.
10H	Indicates that an entry defines a subdirectory. Such an entry is excluded from normal directory searches.
20H	Indicates that the file has been written to and closed. PIP and other file transfer utilities check for this value to determine whether a file was changed since it was last backed up.

You can use the Change File Mode call (43H) to change the read/only, hidden, system, and archive attributes. A file can have any or all of these four attributes in any combination. You cannot change the volume and subdirectory attributes with Change File Mode.

The description of Search for First Entry (11H) contains an explanation of using the attribute byte in file searches.

7.5.4 DOS Disk Transfer Area

Concurrent uses the Disk Transfer Area (DTA) to store data for all file reads and writes performed by a subset of the FCB-oriented function calls. These calls include:

* Search for First Entry (11H)
* Search for Next Entry (12H)
* Sequential Read (14H)
* Sequential Write (15H)
* Random Read (21H)
* Random Write (22H)
* Random Block Read (27H)
* Random Block Write (28H)

When Concurrent gives control to a DOS program, it establishes a default DTA at offset 80H in the program's **Program Segment Prefix** (PSP). See the EXEC call for a description of the PSP.

The default DTA is 128 bytes long. To change the default DTA or establish a new DTA, use the Set Disk Transfer Address call (1AH). Concurrent allows the DTA to be placed in any location within memory, but it must not cross 64KB (segment) boundaries.

Once a DTA is established, Concurrent continues to use that area for all disk operations until a subsequent Set DTA function is performed. You can obtain the current DTA with the Get Disk Transfer Address call (2FH).

Note that under Concurrent, the DOS DTA is equivalent to the **Direct Memory Address** (DMA) buffer.

Disk Reset (0DH)

Flush All File Buffers

Entry Parameters:
Register AH: 0DH

Returned Values:
Register AX: Error code

Disk Reset flushes all file buffers, but does not correctly record disk directory information for those files which were left open and have changed in size.

Your program does not need to call Disk Reset before a disk change when all files written have been closed.

See Table 7-4 for error code definitions.

Select Disk (0EH)

Select a Disk as Default Drive

Entry Parameters:
Register AH: 0EH
DL: Drive vector

Returned Values:
Register AL: Number of drives
AX: Error code

Select Disk first determines whether the drive specified in DL is valid, then selects it as the default drive. Drives are numbered starting with zero, where 0 is A, 1 is B, and so forth.

Interrupt 11H, BIOS Equipment Determination, can also be used to return the number of physical drives; see Section 8.1 "DOS Monitor Calls."

See Table 7-4 for error code definitions.

Open File (0FH)

Open a File for I/O Operations

Entry Parameters:
Register AH: 0FH
DS: Unopened FCB Address - Segment
DX: Unopened FCB Address - Offset

Returned Values:
Register AL: 00H - on success
0FFH - on failure

The Open File call searches the current directory for a specified file. If it finds the file, it opens it, filling in information in the FCB as described below.

On entry, registers DS:DX point to an unopened FCB. Open File searches the current directory for the file named in the FCB, and returns 0FFH in AL if the file is not found. If it is found, Open File returns 00H in AL and fills in appropriate fields in the FCB.

If the drive field is 0, indicating the default drive, Open File changes this field to the actual drive containing the file, where 1 is A, 2 is B, and so forth. This allows the default drive to be changed without interfering with subsequent operations on this file.

Open File sets the current block field to zero. The file record size is set to 80H. Open File sets the file size and date fields from information obtained from the directory.

You should set certain fields in the FCB after Open File has returned but before you request any disk operations. If the file has a record size different from 80H, you must set the record size field (offset 0EH in the FCB). Also, you must set the current record (offset 20H) and/or random record (offset 21H) fields, depending on whether you are doing sequential and/or random reads and writes.

For more detailed information about DOS FCBs, refer to Section 7.5.1, "Standard DOS FCB."

Close File (10H)

Close a File Following Read/Write Operations

Entry Parameters:
Register AH: 10H
DS: Opened FCB Address - Segment
DX: Opened FCB Address - Offset

Returned Values:
Register AL: 00H - On success
0FFH - Directory not updated

The Close File call closes a file following a file write. You must call Close File when finished with the file.

On entry, DS:DX point to an opened FCB. Close File sets AL to 00H to indicate a successful close. If the file is not found in the directory, Close File sets AL to 0FFH.

Search for First Entry (11H)

Find First File to Match Specified FCB

Entry Parameters:
Register AH: 11H
DS: Unopened FCB Address - Segment
DX: Unopened FCB Address - Offset

Returned Values:
Register AL: 00H - Match found
0FFH - No match found

The Search for First Entry call searches for the first filename that matches the filename in the unopened FCB specified in DS:DX. Concurrent searches the current disk directory for the first matching filename. The search includes those filenames containing the question mark character, which matches any character in the same position. If the call does not find a match, it returns 0FFH in AL.

If the call finds a match, it returns 00H in AL and returns information according to whether the FCB pointed to on entry is a normal or extended FCB.

If the search FCB is a normal FCB, Search for First Entry sets the first byte in the calling program's Disk Transfer Area (DTA) to the drive number of the drive containing the matching FCB, where A is 1, B is 2, and so forth. Concurrent copies the matching directory entry to the following 32 bytes in the DTA. On return, the DTA contains an unopened, normal FCB.

If the search FCB is an extended FCB, Search for First Entry sets the first byte in the calling program's DTA to FFH. It then sets the following five bytes to zero, copies the attribute byte and drive code from the search FCB to the following two bytes, then copies the matching directory entry into the following 32 bytes. On return, the calling program's DTA contains an unopened, extended FCB with the same attributes as the search FCB.

If the calling program points to an extended FCB, Search for First Entry bases its search on the contents of the attribute byte in the FCB prefix. Extended FCBs are discussed in Section 7.5.2.

Table 7-9 lists the bits within the DOS attribute byte.

Table 7-9. DOS File Attribute Bits

Bit	Attribute
0	Read/Only
1	Hidden
2	System
3	Volume Label
4	Subdirectory
5	Archive

If the attribute byte is zero, the call finds only normal file entries, which are those that have no attributes set, or have the read/only and/or archive attributes set. Search for First Entry will not return entries for the volume label, subdirectories, or hidden and system files.

If the attribute byte is set for hidden (bit 1 set) or system files (bit 2 set), or for subdirectories (bit 4 set), Search for First Entry searches all normal entries plus all entries matching the specified attributes. To look at all directory entries except the volume label, set bits 1, 2, and 4 in the attribute byte.

If bit 3 is set, Search for First Entry searches only for a volume label.

Search for Next Entry (12H)

Find a Subsequent File Matching the
Specified FCB of a Previous Search Call

Entry Parameters:
Register AH: 12H
DS: Unopened FCB Address - Segment
DX: Unopened FCB Address - Offset

Returned Values:
Register AL: 00H - Match found
0FFH - No match found

After Search for First Entry finds a match to an ambiguous filename, you can call Search for Next Entry to look for the next match. An ambiguous filename is one that contains question mark characters. The Entry parameters and returned values are the same as those for Search for First Entry (11H).

Do not perform any disk operations with the search FCB between Search for First and Search for Next calls, or between two Search for Next Calls, because Concurrent stores information necessary to continue the search in the reserved area of the search FCB. This information would be overwritten by a disk operation that intervened between two search calls.

Delete File (13H)

Delete a Disk File

Entry Parameters:
Register AH: 13H
DS: Unopened FCB Address - Segment
DX: Unopened FCB Address - Offset

Returned Values:
Register AL: 00H - File deleted
0FFH - No matching directory entries

The Delete File call deletes all entries that match the specified filename in the current directory. On entry, DS:DX point to an unopened FCB. The question mark character is allowed in the filename and extension. If no entries match, Delete File returns 0FFH in AL.

Sequential Read (14H)

Sequentially Read Records from a Disk File

Entry Parameters:
Register AH: 14H
DS: Opened FCB Address - Segment
DX: Opened FCB Address - Offset

Returned Values:
Register AL: 00H - On success
01H - End-of-file encountered, no data in record
02H - Insufficient space in DTA to read a single record
03H - End-of-file encountered, partial record read, zero-filled

---

The Sequential Read call reads the record in the file pointed to by the current block (offset 0CH) and current record (offset 20H) fields in the file's FCB. Sequential Read copies the record to the calling program's DTA, then increments the current record field in the FCB. Concurrent determines the length of a file's record from the record size field in the FCB (offset 0EH).

Sequential Read and Sequential Write (15H) increment the FCB's current block field when the current record number field overflows. The range of the current record number field is 0-127. You must initialize the current record number to zero for a read or write that starts from the beginning of the file. For more information on the current block and current record number fields, see Section 7.5.1, "Standard DOS FCB."

Sequential Read returns 01H or 03H in AL if an end-of-file character is encountered. A return of 01H indicates no data in the record; 03H indicates a partial record was read and filled out with zeros.

Sequential Read returns 02H in AL if there was not enough space in the calling program's DTA to read a single record and the read was ended. The call determines this condition when the DTA offset plus the record length exceeds FFFFH. 00H is returned in AL if the read was completed successfully.

Sequential Write (15H)

Sequentially Write Records to a Disk File

Entry Parameters:
- Register AH: 15H
- DS: Opened FCB Address - Segment
- DX: Opened FCB Address - Offset

Returned Values:
- Register AL: 00H - On success
  01H - Disk full
  02H - Insufficient DTA space, operation canceled

The Sequential Write call writes data from the calling program's DTA to the position in the file pointed to by the current block and current record fields in the FCB. After writing each record, Sequential Write increments the current record field in the FCB.

Sequential Write determines the file's record length from the record size field in the FCB. If the record size is smaller than a single sector, Sequential Write buffers data until a sector is filled out, at which time the write is performed.

Sequential Write returns 00H in AL if the write is successfully completed. 01H in AL indicates that the destination disk is full. 02H indicates there is insufficient space in the calling program's DTA (DTA offset plus record length exceeds FFFFH) to hold one record and the operation is ended.

Create File (16H)

Create a Disk File

Entry Parameters:
Register AH: 16H
DS: Unopened FCB Address - Segment
DX: Unopened FCB Address - Offset

Returned Values:
Register AL: 00H - On open
0FFH - No directory entry available

Create File searches the current directory for an entry matching the FCB pointed to by DS:DX. If a matching entry is found, it is reused. If no match is found, Create File searches the directory for an empty entry. When Create File finds an empty directory entry, it initializes the entry to a zero-length file, calls Open File (0FH), and returns 00H in AL.

A file created with the Create File call can be assigned the hidden attribute by using an extended FCB and setting the file's attribute byte to 02H. Section 7.5.3, "DOS File Attribute Byte," describes the attribute byte.

Rename File (17H)

Rename a Disk File

Entry Parameters:
Register AH: 17H
DS: Modified FCB Address - Segment
DX: Modified FCB Address - Offset

Returned Values:
Register AL: 00H - File renamed
0FFH - File not found, or new name already in use

On entry to Rename File, DS:DX point to a modified FCB. The modified FCB has a drive code and filename in the normal position and a second filename starting six bytes after the end of the normal filename field (DS:DX+11H) in what is normally a reserved area.

Rename File changes every filename in the current directory that matches the first filename to the second filename. If question mark characters appear in the second filename, the corresponding positions in the original name are not changed.

Rename File returns 0FFH in AL if no match is found for the first filename or if the second filename is already in use. AL contains 00H if the operation is successful.

Current Disk (19H)

Return Number of Current Default Drive

Entry Parameters:
Register AH: 19H

Returned Values:
Register AL: Current drive number

Current Disk returns the number of the current default drive, where 0 is A, 1 is B, and so forth.

Set Disk Transfer Address (1AH)

Set the Address of the Disk Transfer Area (DTA)

Entry Parameters:
Register AH: 1AH
DS: Disk Transfer Area Address - Segment
DX: Disk Transfer Area Address - Offset

Returned Values:
None

The Set Disk Transfer Area Call sets the address of the **Disk Transfer Area** (DTA) to the value specified in DS:DX. Concurrent does not allow disk transfers to wrap around within the segment or overflow into the next segment.

The space between the offset and the end of the DTA's segment should be large enough to accommodate your largest record. The DOS FCB read and write calls (14H, 15H, 21H, 22H, 27H, and 28H) do not read or write data beyond the end of the segment specified for the DTA, nor do they wrap to the segment's beginning.

If you do not set the DTA, Concurrent uses the default DTA, located at offset 80H in the **Program Segment Prefix** (PSP). You can get the address of the DTA using system call 2FH.

Allocation Table Address (1BH)

Return Information about Default Drive

Entry Parameters:
Register AH: 1BH

Returned Values:
Register DS: FAT ID byte address - segment
BX: FAT ID byte address - offset
DX: Number of clusters
AL: Number of sectors per cluster
CX: Physical sector size

The Allocation Table Address call returns information about the default drive. Upon return, registers DS:BX point to a byte containing the **File Allocation Table** (FAT) identification byte for the default drive. DX contains the number of clusters in the drive, AL contains the sectors per cluster, and CX contains the size of a physical sector.

Allocation Table for Specific Drive (1CH)

Return Information about Specific Drive

Entry Parameters:
- Register AH: 1CH
- DL: Drive Number

Returned Values:
- Register DS: FAT ID byte Address - Segment
- BX: FAT ID byte Address - Offset
- DX: Number of allocation units
- AL: Number of sectors per allocation unit
- CX: Size of a physical sector

This call is identical to call 1BH, except that DL contains a specific drive number. Drive numbers start with zero, where 0 is the default drive, 1 is A, 2 is B, and so forth.

Random Read (21H)

Randomly Read Records from a Disk File

Entry Parameters:
- Register AH: 21H
- DS: Opened FCB Address - Segment
- DX: Opened FCB Address - Offset

Returned Values:
- Register AL:
  - 00H - On success
  - 01H - End-of-file encountered, no data available
  - 02H - Insufficient space in DTA, operation canceled
  - 03H - End-of-file encountered, partial record read, zero-filled

Upon entry to Random Read, DS:DX point to an opened FCB. After Random Read has set the current block (offset 0CH) and current record fields (offset 20H) in the FCB to correspond with the random record field (offset 21H), it reads in the FCB to correspond with the random record field (offset 21H), it reads the indicated record into the calling program's DTA.

Random Read returns 00H in AL when the read is successfully completed. If Random Read encounters an end-of-file indicator, it returns 01H in AL, indicating no more data is available, or 03H, meaning a record was partially read and its remainder filled out with zeros.

Random Read returns 02H in AL when there is not enough space in the DTA to read one record (offset plus record length is greater than 0FFFFH) and the operation was ended.

Random Write (22H)

Randomly Write Records to a Disk File

Entry Parameters:
- Register AH: 22H
- DS: Opened FCB Address - Segment
- DX: Opened FCB Address - Offset

Returned Values:
- Register AL: 00H - On success
  01H - Disk full, write canceled
  02H - Insufficient space in DTA, operation canceled

Upon entry to Random Write, DS:DX point to an opened FCB. After Random Write has set the current block (offset 0CH) and current record fields (offset 20H) in the FCB to correspond with the random record field (offset 21H), it writes the indicated record from the calling program's DTA.

Random Write returns 00H in AL when the write is successfully completed. Random Write returns 01H in AL if the destination disk is full and the operation was canceled. Random Write returns 02H in AL when there is insufficient space in the DTA (offset plus record length is greater than 0FFFFH) to write a single record and the operation was canceled.

File Size (23H)

Return File Size

Entry Parameters:
Register AH: 23H
DS: Unopened FCB Address - Segment
DX: Unopened FCB Address - Offset

Returned Values:
Register AL: 00H - On success
0FFH - No matching entry found

File Size searches the current directory for the first entry that matches the specified FCB pointed to in DS:DX. To receive accurate information from File Size, you must set the record size field (offset 0EH) in the referenced FCB before your program performs the call.

When a matching entry is found, File Size sets the random record field in the FCB to the number of records in the file. File Size counts records in terms of the length of record specified in the FCB's record size field, rounded up. If no matching entry is found, File Size returns 0FFH in AL.

Set Random Record Field (24H)

Set File Address of Random Record Field

Entry Parameters:
Register AH: 24H
DS: Opened FCB Address - Segment
DX: Opened FCB Address - Offset

Returned Values:
None

The Set Random Record Field call sets the random record field (offset 21H) to the file address stored in the current block (offset 0CH) and current record (offset 20H) fields in the FCB.

Random Block Read (27H)

Randomly Read Multiple Records From a Disk File

Entry Parameters:
- Register AH: 27H
- DS: Opened FCB Address - Segment
- DX: Opened FCB Address - Offset
- CX: Record count

Returned Values:
- Register AL: 00H - On success
  - 01H - End-of-file encountered, no data available
  - 02H - Insufficient space in DTA, operation canceled
  - 03H - End-of-file encountered, partial record read, zero-filled
- CX: Number of records read

The Random Block Read call reads the specified number of records from the point in the file specified by the random record field into the calling program's disk transfer area (DTA). Upon entry to Random Block Read, registers DS:DX point to an opened FCB and CX contains a record count that must not be zero. Random Block Read counts records in terms of the record length specified in the record size field (offset 0EH) in the FCB.

If the read was successfully completed, Random Block Read returns 00H in AL. If an end-of-file indicator is reached before all records have been read, Random Block Read returns either 01H or 03H in AL. 01H indicates that the last record is complete; 03H indicates that the last record is a partial record and was filled out with zeros.

Random Block Read reads records into the DTA until the DTA offset plus record length exceeds 0FFFFH. If not all of the specified records have been read, 02H is returned in AL.

In any event, Random Block Read returns the actual number of records read in CX and sets the random record, current block (offset 0CH), and current record (offset 20H) fields to point to the next record in the file; that is, the first record not read.

Random Block Write (28H)

Randomly Write Multiple Records to a Disk File

Entry Parameters:
- Register AH: 28H
- DS: Opened FCB Address - Segment
- DX: Opened FCB Address - Offset
- CX: Record count

Returned Values:
- Register AL: 00H - On success
  01H - Insufficient space on disk, no records written
  02H - Insufficient space in DTA, operation canceled
- CX: Number of records written

Random Block Write writes the specified number of records from the calling program's DTA to the file address specified by the random record field in the opened FCB specified in DS:DX.

Random Block Write returns 00H in AL if the write was successfully completed, or 01H in AL when there is insufficient space on the destination disk and the operation is canceled.

Random Block Write returns 02H in AL when there is insufficient space in the calling program's DTA (offset plus record length is greater than 0FFFFH) to hold one record and the operation is ended.

If CX is zero upon entry, no records are written, but the file is set to the length specified by the random record field. If the new file size is longer or shorter than the file size specified in the FCB (at offset 10H), clusters are allocated or released as appropriate.

Parse Filename (29H)

Parse Specified Filename and Initialize an FCB

Entry Parameters:

Register AH:	29H
DS:	Pointer to command line - Segment
SI:	Pointer to command line - Offset
ES:	Unopened FCB Address - Segment
DI:	Unopened FCB Address - Offset
AL:	Bit map (See below)

Returned Values:

Register DS:	First character following parsed name - Segment
SI:	First character following parsed name - Offset
ES:	First byte of formatted FCB - Segment
DI:	First byte of formatted FCB - Offset
AL:	00H - No wildcard characters used
	01H - Wildcard character in filename or extension
	0FFH - Invalid drive specifier

The Parse Filename call parses an ASCII file specification and prepares an FCB. Upon entry, DS:SI point to a command line to parse and ES:DI point to a buffer to be filled with an unopened FCB.

The command line is parsed for a file name of the form:

```
d:filename.ext
```

If found, Parse Filename creates an unopened FCB for the file at the address pointed to by ES:DI. If no drive specifier is present, the default drive is assumed. If no extension is present, Parse Filename assumes it to be all blanks. If the asterisk (*) character appears in the filename or extension, Parse Filename replaces the asterisk and any remaining characters in the filename or extension with question marks.

The contents of the byte in AL upon entry determine how Parse Filename acts on the specified command line. The bit map for this AL register entry parameter is as follows:

If bit 0 is 1, Parse Filename scans leading separators off the target command line. If bit 0 is 0, no scan-off is performed.

If bit 1 is 1, Parse Filename changes the drive field in the result FCB only if a drive was specified in the command line.

If bit 2 is 1, Parse Filename changes the filename field in the result FCB only if the command line contains a filename.

If bit 3 is 1, Parse Filename changes the extension field in the result FCB only if the command line contains an extension.

Parse Filename ignores bits 4-7.

Filename separators include the following characters:

: colon
. period
; semicolon
, comma
= equal sign
+ plus sign
Tab
Space

Filename terminators include all of the separator characters plus:

\ backslash
< less than
> greater than
| vertical bar
/ slash
" quotation marks
[left square bracket
] right square bracket

Filename terminators also include any control characters.

Parse Filename returns the segment and offset addresses of the first character after the filename in DS and SI, respectively, and returns the address of the first byte of the formatted FCB in ES:DI. If the target command line does not contain a valid filename, Parse Filename returns a blank at ES:DI+1.

If the question mark or asterisk characters appear in the filename or extension, Parse Filename returns 01H in AL. If no wildcard characters appear, AL contains 00H. If the drive specifier is invalid, AL contains 0FFH.

Set/Reset Verify Switch (2EH)

Set Verify Switch for Write Operations

Entry Parameters:
Register AH: 2EH
DL: 00H
AL: 01H - Verify on
00H - Verify off

Returned Values:
None

Concurrent ignores the setting of the Verify flag. This call is supported only for DOS compatibility.

Get Disk Transfer Address (2FH)

Return Address of Current Disk Transfer Area (DTA)

Entry Parameters:
Register AH: 2FH

Returned Values:
Register ES: Current DTA - Segment
BX: Current DTA - Offset

Get Disk Transfer Address returns the address of the current Disk Transfer Area (DTA) in ES:BX. You can set the Disk Transfer Area by calling Set Disk Transfer Address (1AH). The DTA is described in Section 7.5.4.

Get Disk Free Space (36H)

Return Disk Free Space Information

Entry Parameters:
Register AH: 36H
DL: Drive Number

Returned Values:
Register AX: Number of sectors per cluster, 0FFFFH - Invalid drive number
BX: Number of available clusters
DX: Number of clusters in drive
CX: Number of bytes per sector

Get Disk Free Space returns information about the free space on a specified drive. Upon entry, DL contains a drive number, where 0 is the default drive, 1 is A, 2 is B, and so forth.

Get Disk Free Space returns the number of available clusters in BX, the total number of clusters in the drive in DX, the number of bytes per sector in CX, and the number of sectors per cluster in AX.

Get Verify State (54H)

Return Current Value of Verify State

Entry Parameters:
Register AH: 54H

Returned Values:
Register AL: 00H - Verify off
01H - Verify on

Get Verify State returns the value of the verify flag. Get Verify State returns 01H in AL if verify is on, 00H if verify is off.

Note: Concurrent ignores the setting of the Verify flag. This call is supported only for DOS compatibility.

Create a File (03CH)

Create a Disk File

Entry Parameters:
Register AH: 3CH
DS: ASCIIZ string Address - Segment
DX: ASCIIZ string Address - Offset
CX: File attribute

Returned Values:
Register AX: 16-bit handle, if Carry flag clear

Create a File creates a new file or truncates an old file to zero length in preparation for writing. If the file does not exist, this call creates a file in the directory indicated in the ASCIIZ string and gives the file the read/write attribute (see Section 7.5.3, "DOS File Attribute Byte"). Create a File returns the file's handle in AX.

If the carry flag is set, Create a File returns error code 3, 4, or 5 in AX. Error code 5, access denied, indicates that the specified directory is full or a file of the same name exists and is marked read/only. See Table 7-4 for other error code definitions.

Note that you can use Change File Mode (43H) to change the file's attribute.

Open a File Handle (03DH)

Open a File Whose Name Matchs Specified ASCIIZ String

Entry Parameters:
Register AH: 3DH
DS: ASCIIZ string Address - Segment
DX: ASCIIZ string Address - Offset
AL: Access code

Returned Values:
Register AX: Handle or error code

The Open a File Handle call opens any normal or hidden file whose name matches the name and directory location specified in the ASCIIZ string, but does not open files whose names end with a colon.

On entry, the ASCIIZ string contains the drive, path, and filename for the file to be opened. AL contains one of the following access codes:

00H - Open file for reading
01H - Open file for writing
02H - Open file for reading and writing

If the carry flag is set, this call returns error code 2, file not found; error code 4, no file handles (too many open files); error code 5, access denied; or error code 12, invalid access code. If the carry flag is not set, Open a File Handle returns a 16-bit file handle that must be used for subsequent input and output to the file.

Open a File Handle sets the read/write pointer to the first byte of the file and sets the record size of the file to one byte. You can move the file pointer with the Move File Pointer call (42H). You can obtain or change a file's attribute with the Change File Mode call (43H). DOS file attributes are described in Section 7.5.3.

Close a File Handle (03EH)

Close a Specified File Handle

Entry Parameters:
Register AH: 3EH
BX: File handle returned by Open or Create a File

Returned Values:
Register AX: Error code, if Carry flag set

The Close a File Handle call closes the specified file handle, flushes all internal buffers associated with the file, and updates the file's directory. If the carry flag is set, Close a File Handle returns error code 6 in AX to indicate that an illegal file handle was specified.

Note: You should always close a file as soon as you finish using it.

Read from a File or Device (03FH)

Read a Specified Number of Bytes into DMA Buffer

Entry Parameters:
- Register AH: 3FH
- BX: File handle
- DS: DMA address - Segment
- DX: DMA address - Offset
- CX: Number of bytes to be read

Returned Values:
- Register AX: Number of bytes read, if Carry flag not set

The Read from a File or Device call transfers a number of bytes, specified in CX, from a file into the DMA buffer addressed by DS:DX.

On return, if the carry flag is set, AX contains error code 5 or 6. See Table 7-4 for error code definitions.

If the carry flag is not set, AX contains the number of bytes read. If this value is zero, it indicates an attempt to read from the end of the file. If the value in AX is greater than zero, but less than the value specified in CX, it indicates that the program has read up to the end of the file.

Depending on the device being accessed, this call may not read the number of bytes specified in CX. For example, from the keyboard, it reads at most, one line at a time.

Write to a File or Device (040H)

Write Specified Number of Bytes From Internal Buffer

Entry Parameters:
- Register AH: 40H
- BX: File handle
- DS: Address of data to write - Segment
- DX: Address of data to write - Offset
- CX: Number of bytes to write

Returned Values:
- Register AX: Number of bytes written, if Carry flag not set

---

The Write to a File or Device call transfers the number of bytes specified in CX from the buffer pointed to by DS:DX to the file indicated by the file handle in BX.

On return, if the carry flag is set, this call returns error code 5 or 6 in AX. Error code 5 indicates access denied; error code 6 indicates an illegal file handle. If the carry flag is not set, Write to a File or Device returns the actual number of bytes written in AX. If register CX = 0, the file is truncated at the position of the current file pointer.

Your program must compare the number of bytes returned in AX with the number of bytes requested in CX. This call does not return an error if the two values do not match. You should, however, consider this condition an error, since the discrepancy is often caused by a full disk.

Erase a File from Directory (UNLINK) (041H)

Erase Specified File From Directory

Entry Parameters:
Register AH: 41H
DS: ASCIIZ string Address - Segment
DX: ASCIIZ string Address - Offset

Returned Values:
Register AX: Error code, if Carry flag set

This call removes a directory entry associated with a filename. The question mark and asterisk wildcard characters are not allowed in any part of the ASCIIZ string pointed to in DS:DX.

This call does not delete read/only files. To delete a read/only file, you must first use the Change File Mode call (43H) to change the file's attribute. On return, if the carry flag is set, this call returns 2 or 5 in AX to indicate file not found or access denied, respectively.

Move File Read/Write Pointer (042H)

Move Read/Write Pointer to Specified Location

Entry Parameters:
- Register AH: 42H
- CX: Offset to move, in bytes (most significant portion)
- DX: Offset to move, in bytes (least significant portion)
- AL: Move from:
  - 0 - beginning of file
  - 1 - current location
  - 2 - end-of-file

Returned Values:
- Register DX: New pointer location - Segment, if Carry flag not set
- AX: New pointer location - Offset, if Carry flag not set

This call moves the read/write pointer according to the location specified by the value passed in AL. On entry, CX:DX contains the distance to move the pointer. On return, if the carry flag is not set, DX:AX contains the new pointer location. DX contains the most significant part of the value.

If the carry flag is set, this call returns 1 or 6 in AX (see Section 7.3, "DOS System Call Error Return Codes").

Set register AL to one of the following values:

0	Move pointer the number of bytes contained in CX:DX from the beginning of the file.
1	Move pointer to the current location plus the offset contained in CX:DX.
2	Move pointer to the end-of-file plus the specified offset; use this method to determine the file's size.

Change File Mode (043H)

Change a File's Attribute

Entry Parameters:

Register AH:	43H
DS:	Address of ASCIIZ Pathname - Segment
DX:	Address of ASCIIZ Pathname - Offset
CX:	File attribute
AL:	1 - set attribute to CX 0 - return attribute

Returned Values:

Register AX:	None, if Carry flag not set
CX:	File's current attribute, if Carry flag not set and AL contains zero on entry

---

The Change File Mode call changes or returns the file attribute of a file specified by a pointer to an ASCIIZ string. On entry, if AL contains 1, Change File Mode changes the file's attribute to the one specified in CX.

On return, if AL contains zero and the carry flag is not set, this call returns the file's current attribute in CX. The DOS file attribute byte is described in Section 7.5.4.

If the carry flag is set, this call returns 2, 3, or 5 in AX (see Section 7.3, "DOS System Call Error Return Codes,").

Duplicate a File Handle (045H)

Return a Duplicate of an Existing File Handle

Entry Parameters:
Register AH: 45H
BX: File handle

Returned Values:
Register AX: New file handle, if Carry flag not set

This call returns a duplicate of the file handle contained in BX on entry. The duplicate handle refers to the same file in the same directory. If you move the read/write pointer of either handle with the Read (3FH), Write (40H), or Move Pointer (42H) calls, the pointer for the other handle is also moved.

On return, if the carry flag is set, this call returns error code 4 or 6 in AX. Error code 4, no file handles, indicates that the number of open files exceeds system limits; error code 6 indicates an illegal file handle.

Force a Duplicate of a Handle (046H)

Force a Duplicate File Handle to Use
Same Input as an Existing File Handle

Entry Parameters:
Register AH: 46H
BX: Existing file handle
CX: Second file handle

Returned Values:
Register AX: Error code, if carry flag set

This call forces the file handle in CX to refer to the same input stream as the handle contained in BX. If the existing file handle passed in CX refers to an open file, this call closes the file before forcing duplication. If you move the read/write pointer of either handle with the Read (3FH), Write (40H), or Move Pointer (42H) calls, the pointer for the other handle is also moved.

On return, if the carry flag is set, this call returns error code 6 in AX to indicate that an invalid handle has been specified.

Find First Matching File (04EH)

Find First Filename to Match Specified ASCIIZ String

Entry Parameters:
Register AH: 4EH
DS: ASCIIZ string Address - Segment
DX: ASCIIZ string Address - Offset
CX: File attribute

Returned Values:
Register AX: None, if Carry flag not set

The Find First Matching File call finds the first filename that matches the filename in the ASCIIZ string pointed to in DS:DX on entry. The ASCIIZ string contains the drive, path, and filename of the file to be matched. Wildcard characters are allowed in the filename portion of the ASCIIZ string.

This call searches for files according to the attribute contained in CX on entry. The description of Search for First Entry (11H) contains an explanation of how the attribute byte is used for searches.

If a matching file is found, Find First Matching File fills in the calling program's Disk Transfer Area (DTA) as follows:

21 bytes	Reserved for use with subsequent Find Next Matching File (4FH) calls.
1 byte	attribute of matching file
2 bytes	matching file's time of creation
2 bytes	matching file's date of creation
2 bytes	low word of matching file's size
2 bytes	high word of matching file's size
13 bytes	matching file's filename and extension, followed by a byte of zeros. This call removes all blanks from the filename and extension. If the extension is present, it is preceded by a period. The filename returned appears just as you would enter it as a command parameter.

On return, if the carry flag is set, AX contains error code 2 or 18. Error code 2 indicates that the file was not found. Error code 18 indicates that no matching files could be found.

Find Next Matching File (04FH)

Find Next Matching File From Previous Find First Call

Entry Parameters:
Register AH: 4FH

Returned Values:
Register AX: None, if Carry flag not set

The Find Next Matching File call finds the next directory entry matching the filename specified in the previous Find First Matching File (4EH) call. On entry, the DTA must contain the information supplied by a previous Find First File (4EH) call.

If a matching file is found, this call sets the calling program's Disk Transfer Area (DTA) as described under Find First Matching File, above. If no additional matching files are found, this call returns error code 18, no more matching files, in AX.

Rename a File (056H)

Rename a File to Specified ASCIIZ String

Entry Parameters:
Register AH: 56H
DS: ASCIIZ string Address - Segment
DX: ASCIIZ string Address - Offset
ES: ASCIIZ string Address - Segment
DI: ASCIIZ string Address - Offset

Returned Values:
Register AX: Error code, if Carry flag is set

The Rename a File call renames the file specified using the ASCIIZ string pointed to by DS:DX, which contains the drive, path, and name. The ASCIIZ string passed in EX:DI contains the path and new filename.

If a drive is specified in the second ASCIIZ string, it must be the same as that specified in the first string.

On return, if the carry flag is set, AX may contain error code 3, 5, or 17. See Section 7.3, for error code definitions.

Get/Set Time and Date Stamps (057H)

Set or Obtain a File's Date and Time of Creation

Entry Parameters:
- Register AH: 57H
- AL: 00H - Get date/time
  01H - Set date/time
- BX: File handle
- CX: Time to be set if AL = 01H
- DX: Date to be set if AL = 01H

Returned Values:
- Register DX: If getting date, date from handle's internal handle
- CX: If getting time, time from handle's internal handle

This call obtains or sets a file's date and time of creation. If AL contains 01H on entry, this call sets a file's time and date to the values contained in CX and DX. If AL contains zero on entry, this call returns a file's time and date in CX and DX.

Date and time formats are the same as the formats for a DOS directory entry, except that when passed in registers, the bytes are reversed. This means that DH contains the low order byte of the date, DL the high order byte. The DOS file time and date formats are shown in Figures 7-4 and 7-5, respectively.

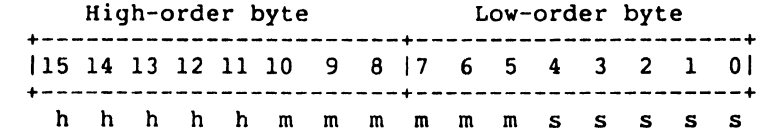


Figure 7-4. DOS File Time Format

h is the binary number of hours, 0 to 23
m is the binary number of hours, 0 to 59
s is the binary number of two-second increments

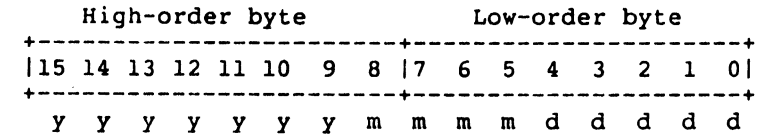


Figure 7-5. DOS File Date Format

y is a binary number of years, 0-119 (1980-2099)
m is the binary number of months, 1 to 12
d is a binary number of days, 1 to 31

On return, if the carry flag is set, AX may contain error code 1 or 6 (see Section 7.3, "DOS System Call Error Return Codes").

Create a Subdirectory - MKDIR (039H)

Create a Subdirectory at the end of the Specified Path

Entry Parameters:
Register AH: 39H
DS: ASCIIZ string Address - Segment
DX: ASCIIZ string Address - Offset

Returned Values:
Register AX: Error code, if Carry flag is set

The ASCIIZ string addressed by registers DS:DX specifies a drive number and directory path name. MKDIR returns to the calling process with a new directory created at the end of the specified path.

MKDIR does not add the new directory to the directory structure if any member of the specified path does not exist. MKDIR returns error code 3 or 5 in register AX. See Table 7-4 for error code definitions.

Remove a Subdirectory - RMDIR (03AH)

Remove a Subdirectory at the end of the Specified Path

Entry Parameters:
Register AH: 3AH
DS: ASCIIZ string Address - Segment
DX: ASCIIZ string Address - Offset

Returned Values:
Register AX: Error code, if Carry flag is set

RMDIR removes the directory specified in the ASCIIZ string addressed by registers DS:DX from the directory structure. RMDIR will not remove the current directory.

RMDIR returns error code 5 if the directory to be removed is not empty, or is being accessed by any process. RMDIR returns error code 3 if the ASCIIZ string specifies an invalid path.

Change Current Directory - CHDIR (03BH)

Set Current Directory as Specified in ASCIIZ String

Entry Parameters:
Register AH: 3BH
DS: ASCIIZ string Address - Segment
DX: ASCIIZ string Address - Offset

Returned Values:
Register AX: Error code, if Carry flag is set

CHDIR sets the current directory as specified in the ASCIIZ string addressed by registers DS:DX.

You can also change floating drives with an ASCIIZ string such as:

```
n:=c:\subdirectory
```

Note: Do not use such a string if DOS compatibility is required.

If any member of the specified path does not exist, CHDIR sets the carry flag and returns error code 3 (Path not found) in register AX.

Get Current Directory (047H)

Return Full Pathname of Current Directory for Specified Drive

Entry Parameters:
Register AH: 47H
DL: Drive number
DS: Path name Address - Segment
SI: Path name Address - Offset

Returned Values:
Register AX: Error code, if Carry flag is set

The Get Current Directory call writes the full path name of the current directory for the drive specified in DL to the 64-byte area of user memory pointed to by DS:SI. Get Current Directory does not include the drive letter or a leading backslash in the path name, and terminates the path name string with 00H.

If DL does not contain a legal drive number, Get Current Directory sets the carry flag and returns error code 15 (Invalid drive specified) in register AX.

Set Vector (025H)

Set Interrupt Vector Table

Entry Parameters:

Register AH:	25H
DS:	Interrupt routine Address - Segment
DX:	Interrupt routine Address - Offset
AL:	Interrupt number (Hex)

The Set Vector call sets the interrupt vector table for the interrupt number passed by the calling process in AL to the address in DS:DX.

Use Get Vector (35H) to obtain the contents of the interrupt vector.

Get DOS Version Number (030H)

Return Major and Minor Version Numbers

Entry Parameters:
Register AH: 30H

Returned Values:
Register AL: Major version number
AH: Minor version number
BX: 00H
CX: 00H

The Get DOS Version Number call returns the high part of the Concurrent version number in register AL and the low part in AH. For Concurrent DOS 86 version 5.0, AL equals 02H and AH equals 0AH. Registers BX and CX are set to 00H.

Get Vector (035H)

Return Address of Interrupt Vector For Specified Interrrupt

Entry Parameters:
Register AH: 35H
AL: Interrupt number (Hex)

Returned Values:
Register ES: Interrupt routine Address - Segment
BX: Interrupt routine Address - Offset

The Get Vector call returns the CS:IP interrupt routine address in registers ES:BX for the interrupt number specified in AL.

Use the Set Vector (25H) call to set interrupt vectors.

Get Country Dependent Information (038H)

Return Settings of Country Dependent Information

Entry Parameters:
Register AH: 38H
DS: Block Address - Segment
DX: Block Address - Offset
AL: 00H

Returned Values:
Register AX: Error code

This call returns the country dependent information shown in Figure 7-7 to the block of memory addressed in registers DS:DX.

The Date format is:

0 M D Y - USA Standard
1 D M Y - European Standard
2 Y M D - Japanese Standard

The Currency format is:

Bit 0 = 0 if currency symbol precedes value
= 1 if currency symbol follows value
Bit 1 = 0 for 0 spaces between currency symbol and value
= 1 for 1 space between currency symbol and value

The Time format is:

Bit 0 = 0 for 12-hour clock
= 1 for 24-hour clock

The Case Map call uses register AL as follows:

Entry AL = ASCII code for character to convert to uppercase
Return AL = ASCII code for uppercase input character

If the carry flag is set, register AX contains an error code on return. See Table 7-4 for error code definitions.

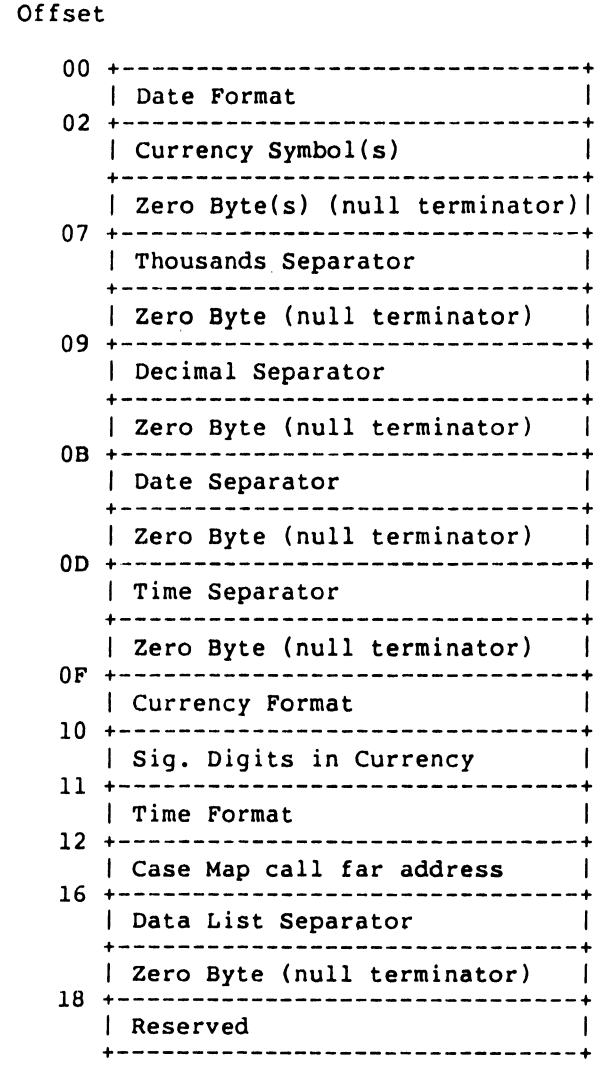


Figure 7-7. Country Dependent Data Return Block

Program Terminate (00H)

Transfer Control to Terminate Address

Entry Parameters:
Register AH: 00H

The Program Terminate call returns the terminate, Ctrl-Break, and critical error exit addresses to the values saved in the Program Segment Prefix (PSP). These are the values saved on entry to the terminating program. See Figure 7-11 for a description of the PSP control block.

Program Terminate flushes all file buffers and transfers control to the terminate address. Note that Concurrent does not properly record the directory information for files that have changed in length and were not previously closed.

Ensure that the CS register contains the segment address of the calling program's PSP before invoking Program Terminate.

Note: You should not use this call in new programs; it is provided only to support DOS pre-version 2.0 programs. The preferred call for this function is EXIT (4CH).

Create New Program Segment (026H)

Create a New program Segment for the Calling Process

Entry Parameters:
Register AH: 26H
DX: Segment number

The Create New Program Segment call copies the entire 100H area from location 0 in the current segment of the calling program into location 0 of the segment whose number is passed in register DX.

This call updates the memory size field (offset 06H) and saves the terminate, Ctrl-Break, and critical error exit addresses for interrupts 22H, 23H, and 24H in the new Program Segment Prefix (PSP) starting at offset 0AH. These addresses are restored from the PSP when the current program terminates.

See the EXEC (4BH) call for a description of the PSP.

Note: You should not use this call in new programs; it is provided only to support DOS pre-version 2.0 programs. The preferred call for this function is EXEC (4BH).

Keep Process (031H)

Terminate the Currently Running Process

Entry Parameters:
Register AH: 31H
AL: Exit code
DX: Memory value - Paragraphs

The Keep Process call terminates the calling process but keeps the number of paragraphs of memory designated by DX. The memory is retained above the start of the current Program Segment Prefix (PSP).

The binary error (or exit) code passed in register AL can be retrieved by the parent process through Get Subprocess Return Code (4DH).

Note: This call should not be used in a Concurrent environment; use a device driver or an RSP instead.

Execute a Program - EXEC (04BH)

Load and Execute a Program or Overlay

Entry Parameters:
- Register AH: 4BH
- AL: 00H - Load and execute program or 03H - Load overlay
- DS: ASCIIZ string address - Segment
- DX: ASCIIZ string address - Offset
- ES: Load parameter block Address - Segment
- BX: Load parameter block Address - Offset

Returned Values:
- Register AX: Error code, if Carry flag is set

The ASCIIZ string addressed by DS:DX must include the name and filetype of the file to be loaded. The drive and path name are optional.

Because Concurrent has previously allocated all available memory to the current program, you must free enough memory for the file to be loaded. Before calling EXEC, use SETBLOCK (4AH) to reduce the allocated memory to the minimum required by the current program.

The format of the **Load Parameter Block** (LPB) pointed to by ES:BX is dependent upon the type of EXEC call being made. See Figures 7-8 and 7-10.

If EXEC returns with the carry flag set, register AX contains one of the following error codes: 1, 2, 7, 8, 10, or 11. See Table 7-4 for error code definitions.

Load and Execute (AL = 00H on Entry)

If AL contains 00H on entry, EXEC creates a **Program Segment Prefix** (PSP) at offset 0 in the program segment and sets its terminate address and Ctrl-Break address fields to the instruction following the EXEC call. EXEC then loads the program indicated in the ASCIIZ string at offset 100H within the program segment and gives it control.

All registers are changed on return from an EXEC load and execute call. You must restore the stack segment, stack pointer, and any other required registers.

The program can return from EXEC in one of five ways:

1. Set AH to 4CH and issue an Interrupt 21H; this is the preferred method.
2. Jump to offset 0 in the Program Segment Prefix.
3. Issue an Interrupt 20H.
4. Set AL to 00H.
5. Call offset 50H in the Program Segment Prefix with AH equal to 4CH.

When control returns to the process that called EXEC (except through call 4CH), ensure that register CS contains the segment address of the Program Segment Prefix. Interrupt vectors 22H, 23H, and 24H are restored from the values in the terminate address, Ctrl-Break address, and critical error exit address fields of the Program Segment Prefix.

Concurrent duplicates the parent process's open files for the child created after a Load and Execute EXEC call. The new process inherits the standard input, output, printer, and auxiliary device definitions from the calling process. The child process also inherits a block of text strings or "environment" from the calling process. If the value of the Environment String Address field in the Load Parameter Block is 0000H, the child process inherits the environment unchanged. Note that the environment must always be located on a paragraph boundary.

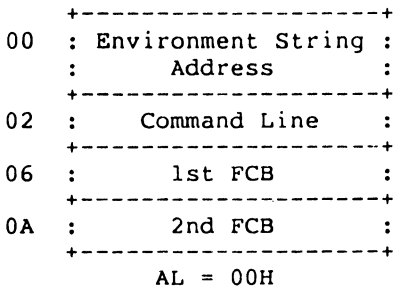


Figure 7-7. EXEC Load and Execute Parameter Block

Table 7-10 defines the fields in the Parameter Block.

Table 7-10. EXEC Load Parameter Block Fields

Field	Definition
Environment String Address	Word address (segment) of the environment passed to PSP.
Command Line	Double-word address of command line to be placed at offset 80H in PSP.
1st FCB	Double-word address of FCB to be placed at offset 5CH in PSP.
2nd FCB	Double-word address of FCB to be placed at offset 6CH in PSP.

Figure 7-8 shows the format of the environment string.

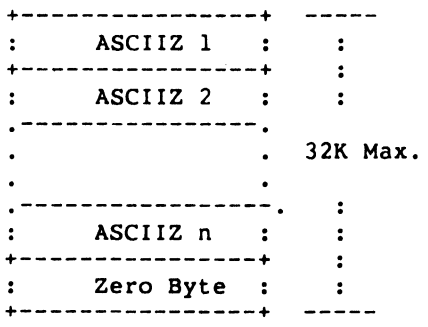


Figure 7-8. DOS Environment String Format

Load Overlay (AL = 03H on Entry)

If the calling process passes 03H in AL on entry, EXEC loads the program indicated in the ASCIIZ string, but does not establish a PSP or begin executing the program. EXEC loads the program in the segment pointed to by the Load Parameter Block depicted in Figure 7-9. Use this type of EXEC call to load program overlays. Figure 7-10 shows the Program Segment Prefix.

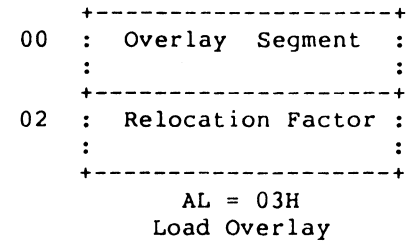


Figure 7-9. EXEC Load Overlay Parameter Block

Table 7-11 defines the fields in the Parameter Block.

Table 7-11. EXEC Load Overlay Parameter Block Fields

Field	Definition
Overlay Segment	Word address of segment in which to load file.
Relocation Factor	Fix-up segment to be applied to the image.

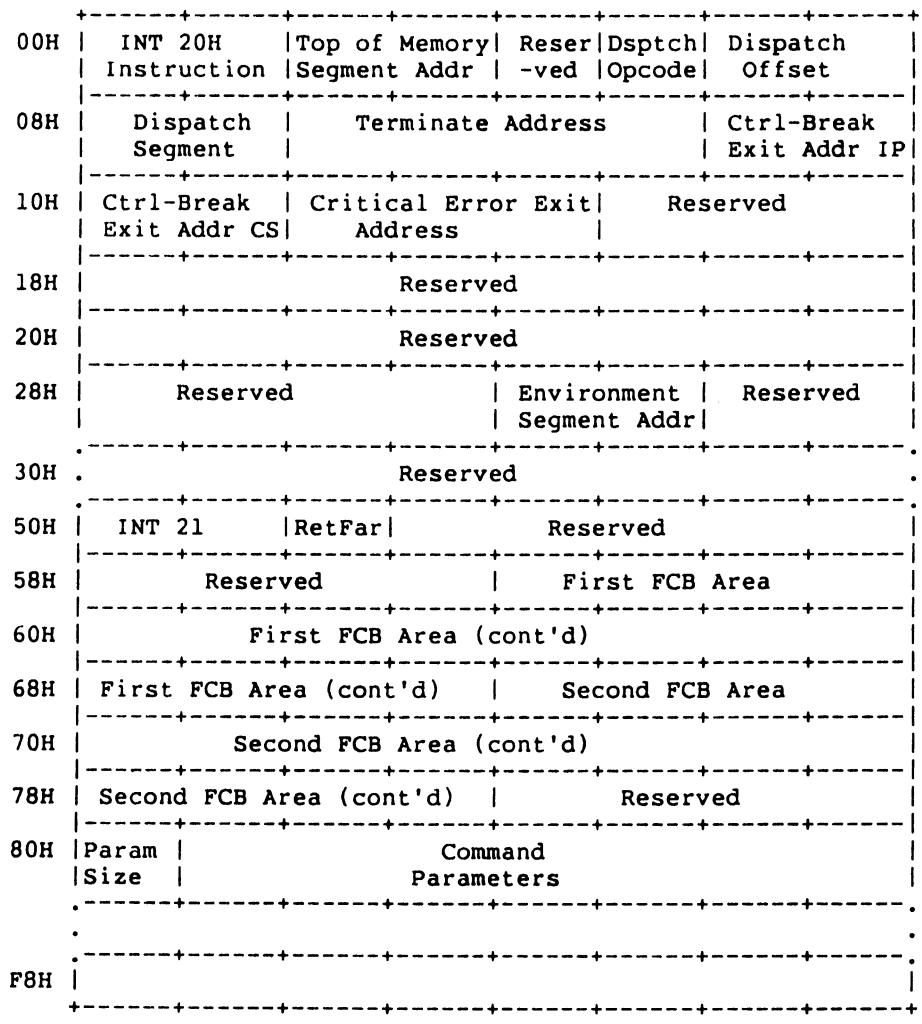


Figure 7-10. DOS Program Segment Prefix

Terminate a Process - EXIT (04CH)

Terminate Current Process and Return Control to Calling Process

Entry Parameters:
Register AH: 4CH
AL: Return code

The Terminate Process call terminates the current process and returns control to the process that invoked EXEC (4BH). EXIT also closes any handles opened by the current process.

The calling process can place a binary return code in register AL before invoking EXIT. This code can then be retrieved by the parent process through Get Subprocess Return Code (4DH).

Get Subprocess Return Code - WAIT (04DH)

Return Completion Code to Calling Process

Entry Parameters:
Register AH: 4DH

Returned Values:
Register AX: Exit code

The Get Subprocess Return Code call returns the binary exit or completion code specified by another process with the Keep Process (31H) or EXIT (4CH) calls. The low byte of register AX contains the exit code specified by the terminating process; the high byte can return one of the following values:

00H - normal termination
01H - Ctrl-Break termination
02H - critical device error
03H - Keep Process (31H) termination

Allocate Memory (048H)

Allocate Specified Number of Paragraphs to Calling Process

Entry Parameters:
Register AH: 48H
BX: Number of paragraphs

Returned Values:
Register AX: Address of block or
AX: Error code, if Carry flag set
BX: Largest available block (on failure)

The Allocate memory call allocates a requested number of paragraphs of memory. On entry, BX contains the number of paragraphs requested. On return, AX:0 points to the allocated memory block. If the allocation fails, Allocate Memory returns the size, in paragraphs, of the largest block of memory available in BX. If the carry flag is set, Allocate Memory returns error code 7 or 8 in AX (see Section 7.3, "DOS System Call Error Return Codes").

Free Allocated Memory (049H)

Free Specified Memory for Reallocation

Entry Parameters:
Register AH: 49H
ES: Block segment

Returned Values:
Register AX: Error code, if Carry flag is set

The Free Allocated Memory call releases the specified memory for reallocation. On entry, ES contains the segment address of the block to be freed.

There are no return parameters. If the carry flag is set, Free Allocated Memory returns error code 7 or 9 in AX (see Section 7.3, "DOS System Call Error Return Codes").

Modify Allocated Memory Blocks - SETBLOCK (04AH)

Modify Memory to Contain Specified Block Size

Entry Parameters:
Register AH: 4AH
BX: Number of paragraphs
ES: Block segment

Returned Values:
Register AX: Error code, if Carry flag is set
BX: Largest available block (On failure), if Carry flag is set

---

This call modifies an allocated memory block to contain a specified block size. On entry, ES contains the segment address of the allocated memory block, and BX contains the new requested block size, in paragraphs. Concurrent attempts to increase or decrease the allocated block according to the contents of BX.

If Concurrent cannot increase the size of the allocated block, SETBLOCK returns the maximum possible block size in BX. If the carry flag is set, SETBLOCK returns error code 7, 8, or 9 in AX. See Table 7-4 for error code definitions.

Get Date (02AH)

Returns Current System Date

Entry Parameters:
Register AH: 2AH

Returned Values:
Register AL: Day of the week
0=Sunday, 1=Monday, 2=Tuesday, 3=Wednesday, etc.
CX: Year (1980-2099)
DH: Month (1-12)
DL: Day (1-31)

The Get Date call returns the day of the week, year, month, and day as recorded by the system clock.

Get Date returns the year in CX, the month in DH, and the day in DL. All returned values are in binary. If the time-of-day clock rolls over to the next day, Concurrent advances the date, taking into account the number of days in each month and leap years.

Set Date (02BH)

Set the System Time and Date

Entry Parameters:
Register AH: 2BH
CX: Year (1980-2099)
DH: Month (1-12)
DL: Day (1-31)

Returned Values:
Register AL: 00H Date was valid
0FFH Date was invalid

The Set Date call sets the system date. On entry, CX must contain a valid year, DH a valid month, and DL a valid day.

Set Date returns 00H in AL if the date is valid and the set operation is successful, or 0FFH if the date is not valid.

Get Time (02CH)

Returns the Current System Time

Entry Parameters:
Register AH: 2CH

Returned Values:
Register CH: Hours (0-23)
CL: Minutes (0-59)
DH: Seconds (0-59)
DL: Hundredths (0-99)

The Get Time call returns the system time in hours, minutes, seconds, and hundredths of a second.

Get Time returns the time in four 8-bit quantities. You can easily convert Get Time's returned values to a printable form or use these values in calculations, such as subtracting one value from another.

Set Time (02DH)

Set the System Time

Entry Parameters:
Register AH: 2DH
CH: Hours (0-23)
CL: Minutes (0-59)
DH: Seconds (0-59)
DL: Hundredths (0-99)

Returned Values:
Register AL: 00H Time is valid
0FFH Time is invalid

The Set Time sets the system time. On entry, CH contains the hours, CL the minutes, DH the seconds, and DL the hundredths of a second.

If any component of the time is invalid, the set operation is terminated and Set Time returns 0FFH in AL. If the time is valid and the time is successfully set, Set Time returns 00H in AL.

End of Section 7

PC DOS Interrupt Support

8.1 PC ROS Monitor Calls

Table 8-1 lists the PC DOS BIOS functions that are available through the PC **Read Only Storage** (ROS) interrupts emulated by Concurrent.

Note: These interrupts are available only in the version of Concurrent called Concurrent PC DOS which has an XIOS written by Digital Research to be compatible with IBM PC DOS. They may not be available in other implementations of Concurrent.

Table 8-1. DOS Monitor Call Interrupts

Hex Number	ROS Interrupt
10	VIDEO_IO
11	EQUIPMENT
12	MEMORY_SIZE_DETERMINE
13	DISKETTE_IO
16	KEYBOARD_IO
17	PRINTER_IO

8.2 DOS Interrupts

Concurrent supports the DOS interrupts listed in Table 8-2.

Table 8-2. DOS Interrupts Supported by Concurrent

DOS Interrupt	Description
20H	Program Terminate
21H	DOS Function Request
22H	Terminate Address
23H	Ctrl-Break Address
24H	Critical Error Exit
25H	Absolute Disk Read
26H	Absolute Disk Write

8.2.1 DOS INT 20H - Program Terminate

Issuing INT 20H restores the terminate (INT 22), Ctrl-Break (INT 23), and critical error exit address (INT 24) fields in the interrupt table to the values they contained when the current program was loaded in the Program Segment Prefix (PSP).

Concurrent flushes all file buffers in response to an INT 20H instruction. If any file that has changed in length is not closed prior to an INT 20H, the directory entries for its length, date, and time will not be correctly recorded (see the Close File (10H) and Close a File Handle (3EH) calls in Section 7). If you want your program to pass an error or completion code before terminating, use Terminate a Process (4CH).

Before issuing an INT 20H, your program must ensure that the CS register contains the segment address of its PSP. See the DOS EXEC (4BH) system call in Section 7 for a description of the PSP control block.

Note: INT 20H is supported for DOS compatibility only; the preferred method for terminating a program is function 4CH "Terminate a Process".

8.2.2 DOS INT 21H - Invoke a DOS System Call

Section 7, "DOS System Calls," describes how to use INT 21H to invoke a DOS system call.

8.2.3 DOS INT 22H - Terminate Address

This interrupt location contains the address to which Concurrent transfers control when the current program terminates. Concurrent copies this address into offset 0AH of the program's Program Segment Prefix when the segment is created.

Do not issue interrupt 22H directly.

8.2.4 DOS INT 23H - Ctrl_Break Address

Concurrent executes interrupt 23H when the user enters Ctrl-Break during standard I/O or standard printer or asynchronous communications operations. If a user enters Ctrl-Break when the current state of Ctrl-Break checking is on (see Ctrl-Break Check 33H), Concurrent issues interrupt 23H on the next DOS system call.

If your Ctrl-Break routine saves all registers, it can continue program execution by ending with an interrupt return instruction (IRET). Concurrent places no restrictions on the routine, including its use of DOS calls, when the registers are saved before it issues an IRET. If your interrupt routine returns with a long return, the carry flag determines whether the program will continue to execute. When the flag is set, the program will end; otherwise, it will continue as with an IRET.

If Ctrl-Break interrupts DOS calls 09H, Print String, or 0AH, Buffered Keyboard Input, Concurrent outputs a Ctrl-C, carriage-return, and linefeed. I/O then continues from the start of the new line if program execution is allowed to continue with an IRET. When the interrupt is issued, all registers are returned to the values they contained during the original DOS system call.

8.2.5 DOS INT 24H - Critical Error Exit Address

Concurrent transfers control to DOS Interrupt 24H when a critical error occurs during the execution of a DOS program. Bit 7 (high-order) of register AH equal to zero indicates a disk error.

INT 24H sets the registers so that if an IRET is executed, Concurrent responds according to one of the following values in AL: 0 = ignore the error; 1 = retry the operation that resulted in the error; 2 = execute DOS INT 23H.

Concurrent will not take an INT 24H exit for errors that occur during a DOS INT 25H or 26H. It takes an INT 24H exit only when the disk error is the result of a DOS INT 21H system call.

When AH bit 7 = 0, register AL contains the number of the drive on which the error occurred (0 corresponds to drive A). Bits 0-2 of AH indicate which operation was in effect and the area of the disk involved in the error. Bits 3-5 of AH indicate valid responses to the disk error. Table 8-3 lists the values assigned to these bits.

Table 8-3. INT 24H Disk Error and Response Indicators

AH Bit	Meaning
0	0 Read operation 1 Write operation
2-1	00 System area 01 FAT 10 Directory 11 Data area
3	0 Fail is not allowed 1 Fail is allowed
4	0 Retry is not allowed 1 Retry is allowed
5	0 Ignore is not allowed 1 Ignore is allowed

Concurrent sets all registers to retry the operation and passes an error code in the low-order byte of register DI. The high-order byte of DI is undefined. Table 8-4 lists these error codes.

Table 8-4. DOS Critical Error Codes

Value of DI (Low-order)	Error
00H	Attempted write on protected diskette
01H	Unit unknown
02H	Drive not ready
03H	Command unknown
04H	CRC error
05H	Bad request structure length
06H	Seek error
07H	Media type unknown
08H	Sector not found
09H	Printer out of paper
0AH	Write failure
0BH	Read failure
0CH	General failure

Concurrent places the information shown in Figure 8-1 on the user stack.

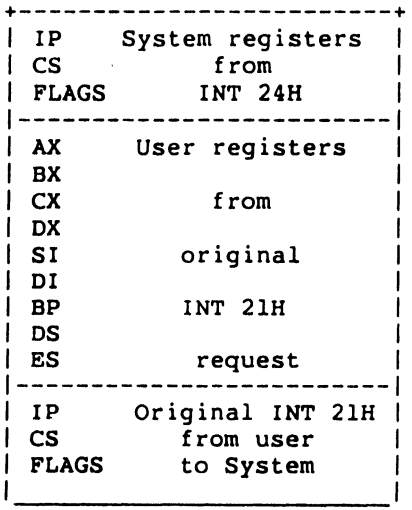


Figure 8-1. User Stack at DOS INT 24H

8.2.6 DOS INT 25H - Absolute Disk Read

DOS Interrupt 25H

Absolute Disk Read

Entry Parameters:
- Register AL: Drive number
- CX: Number of sectors to read
- DX: Beginning logical sector number
- DS: DMA address - Segment
- BX: DMA address - Offset

Returned Values:
- Register AX: Error code, if Carry flag = 1

INT 25H transfers data from memory to disk using the entry parameters. Obtain the logical sector number by numbering each sector sequentially beginning from track 0, head 0, sector 1 (this is logical sector 0) and continuing along the same head, then to the next head until the last sector on the last head of the track is counted.

For example, logical sector 1 is track 0, head 0, sector 2; logical sector 2 is track 0, head 0, sector 3. Numbering continues with sector 1 on head 0 of the next track. Although the sectors are numbered sequentially, they may not be juxtaposed on the disk because of interleaving.

Only the segment registers are preserved by this interrupt call. Concurrent returns to the caller with a Far Return instruction so the original flags remain on the stack and must be explicitly popped off.

Note that if the transfer was successful, the carry flag equals zero. If the carry flag is equal to 1, AX will contain an error code. The error codes returned in AL are the same as those returned in DI by DOS INT 24H. These are listed in Table 8-4, above. The error codes returned in AH are listed in Table 8-5.

Table 8-5. DOS Absolute Disk Read/Write Error Codes

AH Value	Error
00H	General error
02H	Address mark not found
03H	Attempted write on protected diskette
04H	Sector not found
08H	DMA overrun
10H	Bad CRC on read
20H	Controller failure
40H	SEEK failed
80H	Attachment failed

8.2.7 DOS INT 26H - Absolute Disk Write

DOS Interrupt 26H

Absolute Disk Write

Entry Parameters:
- Register AL: Drive number
- CX: Number of sectors to write
- DX: Beginning logical sector number
- DS: DMA address - Segment
- BX: DMA address - Offset

Returned Values:
- Register AX: Error code, if Carry flag = 1

Absolute Disk Write is essentially the same as DOS INT 25H, above, except that it transfers data from memory to disk according to the entry parameters. See DOS INT 25H, Absolute Disk Read, for error code definitions.

End of Section 8

DOS DEVICE DRIVER SUPPORT

This section describes Concurrent's support for DOS device drivers for fixed disks, including Winchester disks, virtual drives in memory (RAM disks), and expanded memory cards. Section 9.1 offers guidelines for writing a DOS fixed-disk driver. Section 9.2 explains how to install DOS drivers under Concurrent.

9.1 Writing a DOS Driver

This subsection presents guidelines for writing a DOS device driver that is installable under Concurrent. Because Concurrent supports a subset of DOS-compatible devices, this section omits certain information on writing drivers that are not applicable to Concurrent.

9.1.1 DOS Driver Format

A DOS device driver follows conventions for COM program files with the significant exception that drivers use an ORG 0 statement (or no ORG statement) rather than ORG 100H. Also, at the beginning of its file, a driver must have a header that identifies it as a device, defines its two entry points, and specifies what type of device it is.

Following the Device Header, a driver contains a series of functions that interface with the driver's device and present data to Concurrent in a specified format. These functions are described in Section 9.1.4.

Be cautious in using any absolute memory references. Concurrent can install a driver in different memory locations at each installation.

9.1.2 DOS Device Header

A DOS driver must begin with a Device Header. Figure 9-1 illustrates the format of this header.

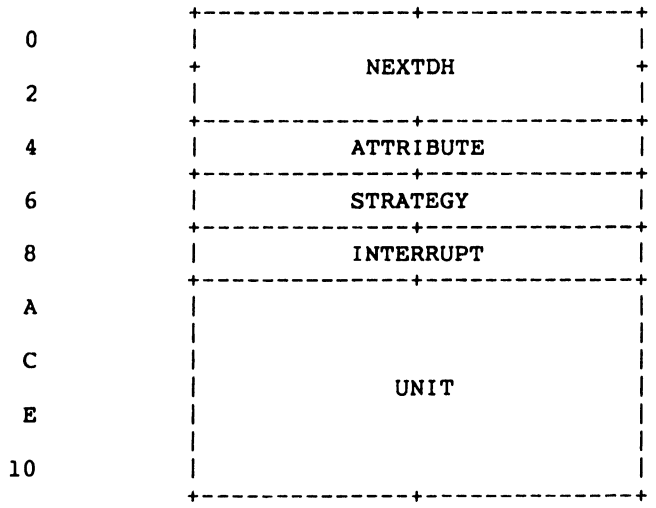


Figure 9-1. DOS Device Header

Table 9-1 defines the fields in the device header.

Table 9-1. DOS Device Header Fields

Field	Description
NEXTDH	This is a double word (offset followed by segment) pointer to the next DOS driver to be loaded. Concurrent sets this field at the time the driver is loaded. This field should be initialized to 0FFFFH,0FFFFH.
ATTRIBUTE	This is a one word bit map that defines a device's characteristics. The bits are used as follows: Bit 15 1 for character devices 0 for block (disk) devices Bit 13 1 non-IBM formatted disk 0 IBM formatted disk

Table 9-1. (Cont'd)

Field	Description
STRATEGY	This field contains a pointer to the strategy entry point. Concurrent calls the strategy routine when it has constructed a **Request Packet**. The Request Packet is discussed in Section 9.1.3. The strategy routine in turn, passes to the interrupt routine a pointer to the Request Packet.
INTERRUPT	Contains a pointer to the interrupt entry point. The interrupt routine processes a Request Packet as passed from strategy routine. The interrupt routine also dispatches I/O requests to the appropriate driver routines, returns requested data to Request Packet, and returns control to Concurrent.
UNIT	Contains the number of units in a disk device. You can place this number in the first byte of this 8-byte field. This is optional, because Concurrent fills in this location with the value returned by a driver's INIT code. If the device is a character device, UNIT contains the device driver's name.

9.1.3 DOS Request Header

When Concurrent receives an I/O request for a DOS device driver, it builds a Request Packet containing information about the request and passes the packet to the driver's strategy entry point. The strategy entry point is called with registers ES and BX pointing to the offset and segment addresses of the Request Packet.

The strategy routine stores this pointer in a queue, then returns to Concurrent. On the strategy routine's return, Concurrent calls the interrupt routine, with no parameters. The interrupt routine obtains the pointer from the queue and performs the requested operation based on the contents of a field within the packet.

The Request Packet consists of a 13-byte header, the Request Header, plus data, in a format dependent on the I/O function that is requested. Figure 9-2 shows the Request Header's format.

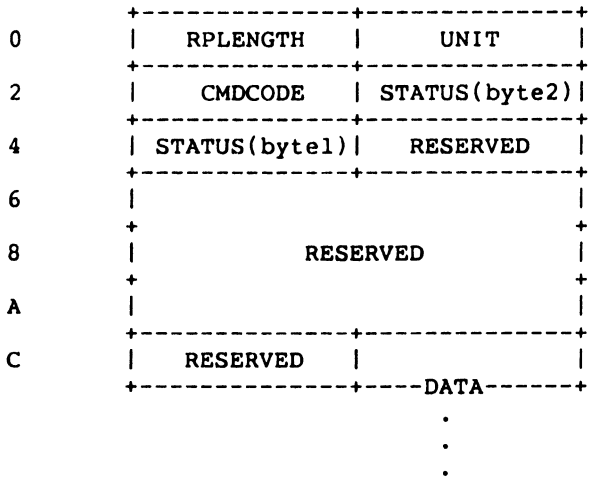


Figure 9-2. Request Header

Table 9-2 defines the fields in the Request Header.

Table 9-2. Fields in Request Header

Field	Description
RPLENGTH	Indicates length, in bytes, of the Request Packet, including header and data.
UNIT	Identifies the unit (logical drive) that is requested. Units are numbered in ascending order from zero.
CMDCODE	Indicates the command code that tells the interrupt routine which driver function is requested. This field can have one of the following values: Code — Function 0 INIT 1 MEDIA CHECK (block devices only) 2 BUILD BPB (block devices only) 3 not used 4 INPUT 5 NON-DESTRUCTIVE INPUT, NO WAIT (character devices only) 6 INPUT STATUS (character devices only) 7 INPUT FLUSH (character devices only) 8 OUTPUT 9 OUTPUT WITH VERIFY 10 OUTPUT STATUS (character devices only) 11 OUTPUT FLUSH (character devices only) 12 not used
STATUS	This word-length field contains the return codes for driver function calls. It is set to zero on entry to the interrupt routine, which sets the field on return to Concurrent. This word is stored in memory, low-order byte first. The status word is a bit map with the following values: Bit 15 — 1 - Error; 0 - No error Bits 14-10 — Reserved Bit 9 — 1 - Device busy (in response to status calls); 0 - Device free

Table 9-2. (Cont'd)

Field	Description
	Bit 8 1 - Operation complete 0 - Operation not complete Bits 7-0 Error code (if bit 15 is set) The following error codes are passed in bits 7-0. Numbers are in hexadecimal notation. 00 Write protect violation 01 Unknown unit 02 Device not ready 03 Unknown command 04 CRC error 05 Bad drive request structure length 06 Seek error 07 Unknown media 08 Sector not found 09 Printer out of paper 0A Write fault 0B Read fault 0C General failure
RESERVED	This is an 8-byte reserved field.
DATA	Contains data pertinent to requested operation presented in a format specific to each driver function.

The following subsection defines the parameter blocks for each function.

9.1.4 DOS Driver Functions

Driver functions fill in parameter blocks that are appended to Request Headers. This section defines the parameter block and other requirements for each function. It also specifies the Request Header command code that the interrupt routine uses to determine which function is being requested.

As stated in Section 9.1.3, each parameter block is preceded by a 13-byte Request Header. Each function must set the status word in the Request Header on its return.

INIT

Command Code: 0

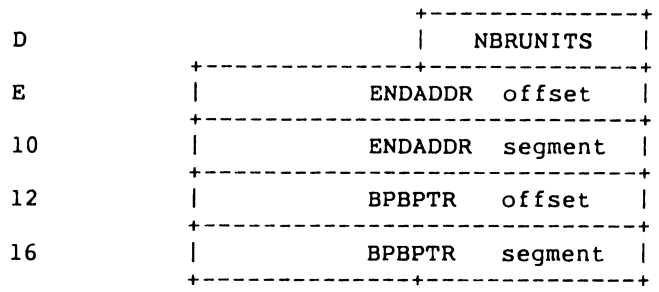


Table 9-3 defines the parameter block fields.

Table 9-3. INIT Parameter Block Fields

Field	Definition
NBRUNITS	Contains the number of units supported by the driver (for block devices only).
ENDADDR	This double-word pointer contains the address of the first byte of free memory after the driver has completed initialization.
BPBPTR	This double-word pointer points to an array of word-length pointers to **BIOS Parameter Blocks** (BPBs). BPBs indicate the default format of each logical drive the driver supports. In the array of word-length pointers, there is a pointer for each unit in the driver. If all units in a driver are the same, the entire array can point to the same BPB to save space.

Figure 9-3 shows the BPB format.

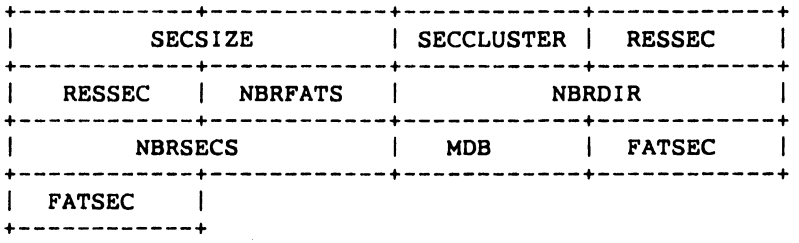


Figure 9-3. BIOS Parameter Block

Table 9-4 defines the fields in the BIOS Parameter Block.

Table 9-4. DOS BIOS Parameter Block Fields

Field	Description
SECSIZE	Contains the number of bytes per sector, a value that must be a multiple of 32.
SECCLUSTER	Contains the number of sectors per cluster, a value that must be a power of two.
RESSEC	Contains the number of sectors reserved for the boot loader.
NBRFATS	Contains the number of FATs on the disk.
NBRDIR	Contains the number of directory entries available in the root directory of the disk.
NBRSEC	Contains the total number of sectors on the disk, including reserved sectors.
MDB	Contains the **Media Descriptor Byte**, which corresponds to the identification byte at the start of the FAT table. The MDB is a bit map that can have the following values: Bit 0 1 two-sided 0 not two-sided Bit 1 1 eight sectors per track 0 not eight sectors Bit 2 1 removable 0 not removable Bits 3-7 must be set to 1
FATSEC	Contains the number of sectors required to contain one copy of the FAT table.

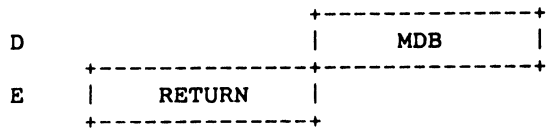
Concurrent calls INIT only at installation time. Place the code for INIT on the high end of the file so that it can be discarded when hardware and software initialization is complete. Concurrent reuses INIT's space by overlaying the next installable DOS driver onto the end of the previous driver, starting at the address specified in the ENDADDR field in INIT's parameter block.

Concurrent allows one driver within a single driver file. A single driver can have multiple units.

The number of units returned by a disk driver determines the logical names for each drive. Concurrent's method of assigning drive letters is explained in Section 9.2.1, "Drive Assignment." The number of units returned by INIT overrides the value in the UNIT field in the Device Header.

MEDIA CHECK

Command Code: 1

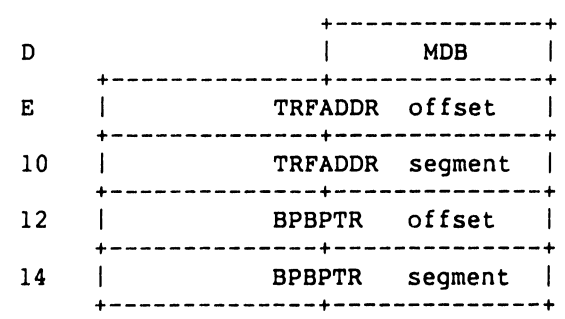


MEDIA CHECK must test the current drive to determine whether the media has been changed since the last disk I/O call. MDB is the current Media Descriptor Byte as passed from Concurrent. In the RETURN field, MEDIA CHECK returns one of the following three results:

-1	Media has been changed.
0	Unsure whether media has been changed.
1	Media has not been changed.

BUILD BIOS PARAMETER BLOCK (BPB)

Command Code: 2



The MDB field, above, is the MDB obtained from Concurrent. TRFADDR is a double-word pointer to a buffer. The setting of the attribute field in the Device Header determines the way the buffer is used. If the disk is IBM format-compatible, the buffer contains the first sector of the FAT table, including the MDB, on the disk. In this case, the driver must not alter the buffer. If the disk is not IBM format-compatible, the buffer specified by TRFADDR can be used as a scratch area.

BPBPTR is a double-word pointer to the BPB returned by the driver after it has determined the density of the disk.

Concurrent calls BUILD BPB under either of the following two conditions:

* If MEDIA CHECK returns "Media changed"
* If MEDIA CHECK returns "Unsure" and there are no buffers with changed data not yet written to disk.

The driver must read the MDB to determine the density of the disk and return it as a BPB.

INPUT or OUTPUT

Command Code: 4 INPUT
8 OUTPUT
9 OUTPUT with verify

All driver input and output functions use the same parameter block, illustrated in Figure 9-4.

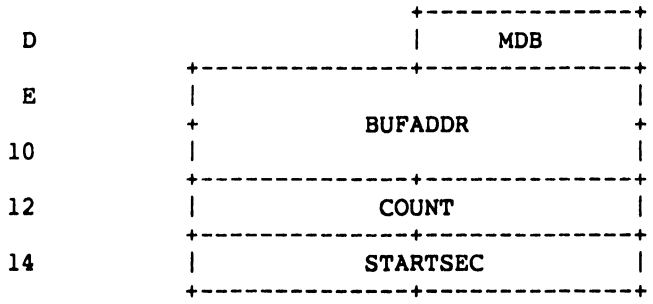


Figure 9-4. Input and Output Parameter Block

Table 9-5 defines the fields in the I/O parameter block.

Table 9-5. Fields in I/O Parameter Block

Field	Description
MDB	This field contains the Media Descriptor Byte for the disk. The driver uses the contents of this field to determine the disk's format.
BUFADDR	This double-word pointer indicates the area of memory that contains the string or sectors if the requested operation is one of the three output functions. If the request is for either of the input functions, BUFADDR is a pointer to the destination area.
COUNT	Contains the number of bytes or sectors to be transferred.
STARTSEC	Contains the starting sector to read or write.

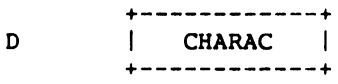
The following is a brief description of each of the input and output functions:

INPUT	Reads a specified number of bytes or sectors from the device. The driver must return an error if the device is not ready for input.
OUTPUT	Sends a specified number of characters or sectors to disk. The driver should return an error if the device is not ready for output.
OUTPUT W/VERIFY	Sends characters or sectors to disk and performs a read after write for verification.

A program using DOS function calls cannot request an input or output operation of more than FFFFH bytes. For this reason, a wrap around in the transfer buffer (pointed to by BUFADDR in the I/O parameter block, above) cannot occur. You can ignore bytes that would have wrapped around in the buffer.

NONDESTRUCTIVE INPUT, NO WAIT

Command Code: 5



CHARAC is a single byte returned from the device. If bit 9, the busy bit, in the status word of the Request Header is set, then the value in CHARAC is meaningless.

The character returned in CHARAC is not removed from the input buffer. This allows Concurrent to look ahead one input character.

STATUS

Command Code: 6 INPUT STATUS
10 OUTPUT STATUS

There is no parameter block for either STATUS call. These functions pass their information back in bit 9, the busy bit, in the status word of the Request Header. STATUS INPUT sets the busy bit to indicate that there is no character in the input buffer. STATUS OUTPUT sets the busy bit to show that the device is not ready for output.

FLUSH

Command Code: 7 INPUT FLUSH
11 OUTPUT FLUSH

These calls have no input or output parameters. The FLUSH functions tell the driver to terminate all pending requests of which it has knowledge.

The FLUSH functions apply to character devices only.

9.2 Installing a DOS Driver

Concurrent loads DOS drivers from disk at boot time in response to commands contained in the ASCII file, CCONFIG.SYS. Concurrent's interface to DOS drivers is contained in the command file named CONFIDD.CMD, (CONFIDD is an acronym for Concurrent Field-installable Device Drivers).

The syntax for a command in CCONFIG.SYS is:

```
FIXED-DEVICE = dvrfil.com

DEVICE = drvfile.com

EMM = drvfile.com
```

where **dvrfil.com** is a file containing the appropriate device driver. FIXED-DEVICE is a fixed (hard) disk. DEVICE is a removable block device, or any character device other than the standard DOS logical names: CON, AUX, PRN, or CLOCK. EMM is an expanded memory hardware device that uses the Intel/Lotus or AST specification.

Note: In the absence of a drive specifier, the device driver file must reside on the drive from which Concurrent boots.

When Concurrent boots, TMP0 (Terminal Message Process for screen 0) searches for the file CCONFIG.SYS. If found, TMP0 then executes the command file CONFIDD.CMD, which installs each device driver named in CCONFIG.SYS, and then returns control to the system console.

9.2.1 Memory Requirements

During installation, each block device (fixed or removable drive) requires some memory allocation from Concurrent's memory pool. For some device drivers, the memory pool can be exhausted, in which case the installation will fail. Should this occur, perform the following steps:

1. Run the SETUP utility and select F2, the "Reserve System Space" menu.
2. Increase the amount of memory reserved for device drivers.
3. Update the CCPM.SYS file on the boot disk, then reboot Concurrent.

Repeat steps 1-3 until the amount of reserved memory is sufficient for successful installation of all the device drivers.

9.2.2 Drive Assignment

The number of units returned by a disk driver in its Device Header determines the logical names for each drive. Concurrent assigns drive letters starting with the first available free drive, assigning only unused drive letters to subsequent units. Permanently linked drivers do not change their drive assignments. Concurrent initializes DOS drivers in the order in which they are read from CCONFIG.SYS.

For example, on a machine with one permanent floppy disk and one permanent hard disk drive with two partitions, Concurrent assigns drive letters as follows:

A: floppy drive
C: DOS partition on hard disk
D: Concurrent partition on hard disk

In initializing a DOS driver with four units, Concurrent assigns the units starting with the first available drive, drive B, then skips to the next available drives, E, F, and G, for the remaining three units. Additional DOS drivers in CONFIG.SYS would be assigned drive letters starting with H.

To determine the number of units in a driver, Concurrent uses the value returned by the driver's INIT function, not the value in the UNIT field of the Device Header.

Concurrent's method of loading DOS drivers allows each driver's drives to be completely installed before the next driver is loaded. Concurrent can then access and load the next DOS driver from the previous driver's drives.

End of Section 9

WINDOW MANAGEMENT

This section describes how an application program can perform efficient console output while working within Concurrent's windows and summarizes the use of Concurrent's window management primitives for the IBM® Personal Computer, PC/XT, and PC/AT. Concurrent's window management primitives are implemented in the hardware-specific XIOS. Also described in this section are the Concurrent window management escape sequences.

10.1 Virtual Consoles

In Concurrent, each process must own a **virtual console** to perform console output. The virtual console is actually an area of memory of the same size and format as the physical console. The virtual console buffer saves the output of a process while another process is using all or part of the physical console.

Concurrent's XIOS contains sophisticated window management functions that ensure the virtual console images are updated and that the physical console displays the appropriate portions of each virtual console. Windowed portions of each virtual console can all be shown on the physical console at the same time.

10.2 Virtual Console Output

Because different portions of a virtual console can be displayed on the physical console in almost any location, it is difficult to write directly to the physical console and still work in windows. This section describes a mechanism that allows a program to update its virtual console buffer and then use the XIOS to update the physical console with the appropriate portion of the virtual console.

This method of handling console output uses some non-standard XIOS entry points that allow you to write directly to the virtual console buffer and still work in windows.

Note that although the mechanism is extremely efficient and exploits the full capabilities of the IBM PC character map, it is not portable. It requires both a complex interface to the XIOS, and the ability to link with either an assembly language routine or the escape sequences described in Section 10.4.

The method described here is used by the program, SAMPLE.C, on the distribution disk. SAMPLE.C links with WWCALL.A86, an assembly language routine also contained on the distribution disk.

There are six general steps you must perform to use this method of handling console output:

1. Call the C_GET BDOS function to find your default virtual console number:

```
vc_number = __BDOS(0x99,0x0);
```

2. Make a special XIOS call to find the address of the virtual console screen buffer that corresponds to your default virtual console:

```
WM_PK(0,vc_number,VS_VC_SEG,&vc_segment);
```

3. Make an XIOS call (IO_SWITCH) to ensure that what is currently on the physical console is synchronized with your virtual console screen buffer. The contents of the buffer and the physical console might not correspond if another program has sent output directly to the physical console or if the virtual console window is full screen, on top (switched in), and positioned at 1,1 on the physical console:

```
WORD          ax,bx,cx,dx;   /* (declaration) */
ax = 7;                      /* IO_SWITCH (7H) */
bx = cx = 0;
dx = vc_number;
IO_CALL(&ax,&bx,&cx,&dx);  /* C entry in WWCALL.A86 */
```

4. Make a special XIOS call to gain ownership of an internal mutual exclusion queue (semaphore) that protects virtual console buffers from being updated by another process:

```
WM_MXQ(0);
```

5. You can now update your virtual console buffer directly. After updating the buffer, immediately make a special XIOS call that frees the internal virtual console semaphore. No other XIOS window management functions that involve your virtual console can occur while you have the semaphore that protects your virtual console. The following call frees the semaphore:

```
WM_MXQ(1);
```

6. Use the following series of special XIOS calls to update the physical console with the current contents of the virtual console:

```
WORD          ax,bx,cx,dx;   /* (declaration) */
WM_PK(0,vc_number,2, &cx);   /* top left */
WM_PK(0,vc_number,4, &bx);   /* bottom right */
ax = 20;                     /* WW_NEW_WINDOW (14H) */
bx = vc_number;
IO_CALL(&ax,&bx,&cx,&dx);  /* C entry in WWCALL.A86 */
```

WM_PK, IO_CALL, and WM_MXQ are contained in WWCALL.A86. WM_PK is the window manager peek/poke routine (0 = peek, 1 = poke). IO_CALL is the C language interface routine. WM_MXQ obtains or releases the internal virtual console semaphore (0 = get, 1 = release). See Section 4, "Console I/O Functions," in the System Guide for a description of the IO_SWITCH (07H) XIOS call.

Repeat steps 4, 5, and 6 as often as necesssary. You must perform step 3 everytime you combine a call to a BDOS console output function with a direct update of your virtual console buffer. Note that the XIOS does not keep the virtual and physical consoles synchronized when a full screen is running in the foreground on a monochrome display. This allows those programs that write directly to the display to work as long as they run only in full screen, foreground.

The XIOS calls that allow you to directly control Concurrent's windows are described in the next section, "XIOS Window Management Calls."

10.3 XIOS Window Management Calls

In addition to handling console output so that it works within windows, you can control the placement, size, scrolling, tracking, and ordering (what is on top) of the windows. This direct interaction with the window management primitives can be very useful, but it is also very XIOS-dependent and, therefore, not portable.

This section briefly describes the XIOS window management calls and summarizes their entry parameters and return values. Also described in this section are two important window management data structures: the **Virtual Console Structure** and the **Window Data Block**.

Note that the window management escape sequences described in Section 10.4 provide a subset of the XIOS window management functions. These escape sequences do not require applications to link with assembly language routines in order to make XIOS window management calls.

There are nine XIOS window management calls. Their names and functions are listed in Table 10-1.

Table 10-1. XIOS Window Functions

Routine	Function
WW_POINTER	This call returns a pointer to either the Virtual Console Structure or the Window Data Block. These data structures, described in Tables 10-3 and 10-4, respectively, indicate the size and position of a window, a window's position in relation to other windows, and other useful information.
WW_KEY	WW_KEY supports the WMENU utility.
WW_STATLINE	This call is used by the window manager to control its status display. An application may use this call to write to the status line area of the physical console.
WW_IM_HERE	This call allows an application to switch a new window to the foreground.
WW_NEW_WINDOW	With this call, an application can place and size any window on the screen. By passing the current window boundaries as the new window boundaries, an application can also use this call to write the virtual console buffers to the physical console.
WW_CURSOR_VIEW	This call sets cursor track mode and viewpoint. An application can use it to scroll the window over the console buffer, displaying a different portion of the 80 x 24 field.
WW_WRAP_CLOUMN	An application can use this call to set the column for automatic wrap-around. This prevents characters from being lost outside the window during simple console output calls.
WW_FULL_WINDOW	This call toggles the current top window between full screen and its previous definition.
WW_SWITCH_DISPLAY	In a two-monitor system, this call moves a window from one physical console to another, clears both screens, and updates all windows.

The XIOS routines listed in Table 10-1 are non-standard or "backdoor" in the sense that they cannot be called using the standard Function 50 XIOS calling convention.

These routines are called through a far call to the XIOS entry point using the standard XIOS segment register conventions, specifically: DS = SYSDAT and ES = UDA.

At entry, each routine is called with a function code in AL, and various parameters in BX, CX, and DX. Table 10-2 summarizes the XIOS window management calls in terms of register content at entry and exit. For more complete information about the XIOS window management calls, see Section 4 of the System Guide.

Table 10-2. XIOS Window Management Call Summary

Function	Input Parameters	Returned Values
WW_POINTER	AL = 10H	
	DL = vc number	AX = .vc struct
	DL = FF for window info	AX = .window data blk
WW_KEY	AL = 11H	AL = char/00
	CL = FF for input/status	AL = FF/00
	CL = FE for status only	AL = char
	CL < FF wait for input	AH = key type
		00 - regular
		FF - special
WW_STATLINE	AL = 12H	none
	CX:DX = .word string	
WW_IM_HERE	AL = 13H	none
	CL = Window Manager state	
	0 - not resident	
	1 - resident, not active	
	2 - resident and active	
	3 - switch vc to top	
	DL = vc number to switch	
	DL = FF - no console switch	
WW_NEW_WINDOW	AL = 14H	none
	DL = vc number	
	CH = top left row	
	CL = top left column	
	BH = bottom left row	
	BL = bottom left column	

Table 10-2. (Cont'd)

Function	Input Parameters	Returned Values
WW_CURSOR_VIEW	AL = 15H DL = vc number DH = tracking mode 0 - fixed window 1 - track scrolling cursor CH = top left row CL = top left column	none
WW_WRAP_COLUMN	AL = 16H CL = Wrap column number DL = vc number	none
WW_FULL_WINDOW	AL = 17H DL = vc number	none
WW_SWITCH_DISPLAY	AL = 18H CL = mode 0 - monochrome 1 - color DL = vc number	none

Concurrent maintains a virtual console data structure for each virtual console in the system. WW_POINTER returns a pointer to the virtual console data structure that corresponds with the virtual console number passed as an entry parameter. Structure members include cursor position, window position, tracking mode, and console buffer segment. Table 10-3 describes the virtual console data structure, which is returned by WW_POINTER when DL = vc number.

Table 10-3. Virtual Console Structure Definition

Field	Offset	Description
vs_cursor	word ptr 00	cursor row,col position in vc image
vs_top_left	word ptr 02	inside top,left corner of vc window
vs_bot_right	word ptr 04	inside bottom,right corner of window
vs_old_t_l	word ptr 06	saves the previous value of top_left
vs_old_b_r	word ptr 08	saves the previous value of bot_right
vs_crt_size	word ptr 10	total number of rows, cols in vc image
vs_win_size	word ptr 12	used by WMENU window manager to save the size of the user's window; this is not updated by the XIOS
vs_view_point	word ptr 14	top_left image corner to top_left win
vs_rows	word ptr 16	number of rows in current vc window
vs_cols	word ptr 18	number of columns in current window
vs_correct	word ptr 20	window tracking correction factor
vs_vc_seg	word ptr 22	segment address of vc console image
vs_crt_seg	word ptr 24	segment address of crt memory
vs_list_ptr	word ptr 26	points to the start of row update list
vs_attrib	byte ptr 28	current vc char attribute
vs_mode	byte ptr 29	cursor on/off and wrap on/off
vs_cur_track	byte ptr 30	window fixed or tracking scroll
vs_width	byte ptr 31	column to wrap around
vs_number	byte ptr 32	copy of the vc number
vs_bit	byte ptr 33	vc num = bit pos

Table 10-3. (Cont'd)

Field	Offset	Description
vs_save_cursor	word ptr 34	ESC code save/restore cursor
vs_vector	word ptr 36	conout state machine vector
vs_xlat	word ptr 38	mono/color xlat table
vs_qpb	word ptr 40	reserved
vs_true_view	word ptr 42	corrected view point
vs_cur_type	word ptr 44	mono or color
vs_mxsemaphore	byte ptr 52	internal XIOS semaphore

When DL = FFH on entry to WW_POINTER, the call returns a pointer to the Window Data Block. This structure is described in Table 10-4.

Table 10-4. Window Data Block Definition

Field	Offset	Description
ww_im_here	byte ptr 0	manager process state variable 0 = manager not resident 1 = manager resident but not active 2 = manager resident and active
nvc	byte ptr 1	number of virtual consoles
priority	byte ptr 2	a list of vc numbers (nvc bytes long) from the back window to the front window

Windows are specified by their inside corners. Rows are from 0 to 23, columns from 0 to 79. If the new size is the same as the the old size, then the physical image is updated from the virtual screen buffer. This ensures that changes to the virtual screen buffer are correctly displayed on the physical console.

10.4 Escape Sequences

Window management escape sequences provide a limited subset of the XIOS window management functionality. Use the escape sequences when you cannot or do not want to link your application with assembly language routines that make backdoor XIOS calls. These escape sequences all have the same basic format.

In general, they are a list of the bytes to be stored in AL, AH, BL, BH, CL, CH, DL, DH before making an XIOS call. This escape sequence mechanism supports the XIOS calls listed in Table 10-5.

Table 10-5. XIOS Calls for Escape Sequences

XIOS Call	Hex Code	Function
io_switch	7	Switches screen to top but does not give it the keyboard--if screen is already on top, updates the vc buffer with what is currently on the physical console
io_statline	8	Displays 80 character status line
ww_statline	12	New status line call with attributes
ww_im_here	13	Sets the window manager process state and changes which console is on top
ww_new_window	14	Sets a new console window
ww_cursor_view	15	Sets cursor track mode and viewpoint
ww_wrap_column	16	Sets the column for auto wrap-around
ww_full_window	17	Toggles indicated vc from full screen to not full screen
ww_switch_display	18	Sets the console for the vc

The escape sequences use a parameter list of registers to access the XIOS calls described in Table 10-5. There are always ten bytes in the escape sequence. The first two bytes contain the terminal-oriented escape code. The last eight bytes must contain the values the registers are to contain before an XIOS call. The exact escape sequence format is as follows:

```
ESC ! al ah bl bh cl ch dl dh
```

Note: If you use these escape sequences in an application, you sacrifice portability for the ability to use windows.

End of Section 10

ECHO.A86 - SAMPLE RSP

This appendix contains the code listing for the Resident System Process, ECHO.

Listing A-1. ECHO.A86

```
;
;       ECHO - Resident System Process
;       Print Command tail to console
;
;
;       DEFINITIONS
;

ccpmint         equ     224     ;ccpm entry interrupt
c_writestr      equ     9       ;print string
c_detach        equ     147     ;detach console
c_set           equ     148     ;set default console
q_make          equ     134     ;create queue
q_open          equ     135     ;open queue
q_read          equ     137     ;read queue
q_write         equ     139     ;write queue
p_priority      equ     145     ;set priority
pdlen           equ     48      ;length of PD
p_cns           equ     byte ptr 020h   ;default cns
p_disk          equ     byte ptr 012h   ;default disk
p_user          equ     byte ptr 013h   ;default user
p_list          equ     byte ptr 024h   ;default list
ps_run          equ     0               ;PD run status
pf_keep         equ     2               ;PD nokill flag
rsp_top         equ     0       ;rsp offset
rsp_pd          equ     010h    ;PD offset
rsp_uda         equ     040h    ;UDA offset
rsp_bottom      equ     140h    ;end rsp header
qf_rsp          equ     08h     ;queue RSP flag

;
;       CODE SEGMENT
;
        CSEG
        org 0
```

Listing A-1. (Cont'd)

```
ccpm:   int ccpmint
        ret
main:           ;create ECHO queue
        mov cl,q_make ! mov dx,offset qd
        call ccpm
                ;open ECHO queue
        mov cl,q_open ! mov dx,offset qpb
        call ccpm
                ;set priority to normal
        mov cl,p_priority ! mov dx,200
        call ccpm

                ;ES points to SYSDAT
        mov es,sdatseg

loop:   ;forever
                ;read cmdtail from queue
        mov cl,q_read ! mov dx,offset qpb
        call ccpm

                ;set default values from PD
        mov bx,pdadr
;       mov dl,es:p_disk[bx]    ;p_disk=0-15
;       inc dl ! mov disk,dl    ;make disk=1-16
;       mov dl,es:p_user[bx]
;       mov user,dl
;       mov dl,es:p_list[bx]
;       mov list,dl
        mov dl,es:p_cns[bx]
        mov console,dl

                ;set default console
;       mov dl,console
        mov cl,C_SET ! call ccpm

                ;scan cmdtail and look for '$' or 0.
                ;when found, replace w/ cr,lf,'$'

        lea bx,cmdtail ! mov al,'$' ! mov ah,0
        mov dx,bx ! add dx,131
```

Listing A-1. (Cont'd)

```
nextchar:
        cmp bx,dx ! ja endcmd
        cmp [bx],al ! je endcmd
        cmp [bx],ah ! je endcmd
            inc bx ! jmps nextchar

endcmd:
        mov byte ptr [bx],13
        mov byte ptr 1[bx],10
        mov byte ptr 2[bx],'$'

                ;write command tail

        lea dx,cmdtail ! mov cl,C_WRITESTR
        call ccpm
                ;detach console
        mov dl,console
        mov cl,c_detach ! call ccpm
                ;done, get next command
        jmps loop

;
;       DATA SEGMENT
;
        DSEG
        org     rsp_top

sdatseg         dw      0,0,0
                dw      0,0,0
                dw      0,0

        org     rsp_pd

pd              dw      0,0             ; link,thread
                db      ps_run          ; status
                db      190             ; priority
                dw      pf_keep         ; flags
                db      'ECHO    '      ; name
                dw      offset uda/10h  ; uda seg
                db      0,0             ; disk,user
                db      0,0             ; load dsk,usr
                dw      0               ; mem
                dw      0,0             ; dvract,wait
```

Listing A-1. (Cont'd)

```
                db      0,0
                dw      0
                db      0                       ; console
                db      0,0,0
                db      0                       ; list
                db      0,0,0
                dw      0,0,0,0

        org     rsp_uda

uda             dw      0,offset dma,0,0        ;0
                dw      0,0,0,0
                dw      0,0,0,0                 ;10h
                dw      0,0,0,0
                dw      0,0,0,0                 ;20h
                dw      0,0,0,0
                dw      0,0,offset stack_tos,0  ;30h
                dw      0,0,0,0
                dw      0,0,0,0                 ;40h
                dw      0,0,0,0
                dw      0,0,0,0                 ;50h
                dw      0,0,0,0
                dw      0,0,0,0                 ;60h

        org     rsp_bottom

qbuf            rb      131             ;Queue buffer

qd              dw      0               ;link
                db      0,0             ;net,org
                dw      qf_rsp          ;flags
                db      'ECHO    '      ;name
                dw      131             ;msglen
                dw      1               ;nmsgs
                dw      0,0             ;dq,nq
                dw      0,0             ;msgcnt,msgout
                dw      offset qbuf     ;buffer addr.

dma             rb      128

stack           dw      0ccccch,0ccccch,0ccccch
                dw      0ccccch,0ccccch,0ccccch
                dw      0ccccch,0ccccch,0ccccch
                dw      0ccccch,0ccccch,0ccccch
                dw      0ccccch,0ccccch,0ccccch
```

Listing A-1. (Cont'd)

```
stack_tos       dw      offset main     ; start offset
                dw      0               ; start seg
                dw      0               ; init flags

pdadr           rw      1               ; QPB Buffer
cmdtail         rb      129             ; starts here
                db      13,10,'$'

qpb             db      0,0             ;must be zero
                dw      0               ;queue ID
                dw      1               ;nmsgs
                dw      offset pdadr    ;buffer addr.
                db      'ECHO    '      ;name to open

console         db      0
;disk          db      0
;user          db      0
;list          db      0

        end
```

End of Appendix A

8087 Exception Handling

This appendix includes an example of an 8087 interrupt handling routine to demonstrate the requirements for using the 8087 processor. Refer to Intel's iAPX 86,88 User's Manual for a description of 8087 exception handling in the section on "8087 Numeric Data Processor".

In order to guarantee the data integrity for each 8087 process in the multitasking environment, any user-defined exception handler must adhere to a minimum sequence of steps within the exception handler:

1. Save the 8086 environment of the 8086-running process.
2. Save the environment of the 8087-running process. The OWNER_8087 field in SYSDAT will contain the offset of the 8087-running process (see description of SYSDAT in Section 6).
3. Clear the 8087 interrupt request bit in the status word.
4. Disable the 8087 interrupts.
5. Clear the PIC interrupt (this instruction is hardware-dependent).
6. At this point, you might want to modify the 8087 environment image saved in step 2 above.
7. Before enabling the 8086 interrupts, restore the 8087 environment with its status word's interrupt request bit cleared. If the environment is not restored before 8086 interrupts are enabled, and an interrupt occurs (like a tick), a different 8087 process can gain control of the 8087 and swap in its 8087 context. On a second interrupt, or on an IRET instruction, the 8086-running process that happened to be executing the exception handler code is brought back into 8086 context and writes over the new 8087 context.

The user program, which uses its own exception handler, must replace the system's interrupt vector with its own. Once this is done, Concurrent swaps this vector into memory every time the program comes back into 8087 context. The address of the interrupt vector is in the SYSDAT table at offset A0H.

The default exception handler aborts those 8087 programs that have enabled 8087 interrupts and that generate a severe error (such as stack underrun, divide by zero, and so forth). Any other errors are ignored by the default exception handler.

Listing B-1. 8087 Exception Handling

```
;=======
ndpint:          ; 8087 interrupt routine
;=======
; This exception handler is non-specific and
: is meant as an example
; default. It is assumed that if the 8087
; programmer has enabled 8087
; interrupts and has specified exception flags
; in the control word, then
; the programmer has also included an
; exception handler to take
; specific actions within the program
; before continuing in the 8087.
; This handler will ignore non-severe
; errors (overflow,etc) and will
; terminate processes with severe errors
; (divide by zero,stack violation).

  push ds                 ; SAVE CURRENT DATA SEGMENT
  mov ds,sysdat           ; GET XIOS DATA SEGMENT
  mov ndp_ssreg,ss        ; DO STACK SWITCH FOR 8086 ENV
  mov ndp_spreg,sp        ; SAVE
  mov ss,sysdat
  mov sp,offset ndp_tos   ; SAVE THE 8086 REGISTERS
  push ax! push bx
  push cx! push dx
  push di! push si
  push bp! push es
  mov es,sysdat           ; NOW SAVE THE 8087 ENV
  FNSTENV env_8087        ; SAVE 8087 PROCESS INFO
  FWAIT
  FNCLEX                  ; CLEAR ITS INT REQUEST BIT
  xor ax,ax               ; WAIT
  FNDISI                  ; DISABLE ITS INTERRUPTS
  mov al,020h             ; SEND 2 INTERRUPT ACKNOWLEDGES,
  out 060h,al             ; 1 FOR MASTER PIC, 1 FOR SLAVE
  mov al,020h
  out 058h,al
  call in_8087            ; IN_8087 CHECKS THE 8087 ERROR
                          ; CONDITION. IF ERROR IS SEVERE,
                          ; IT WILL ABORT, ELSE IT WILL
                          ; RETURN WITH NO CHANGES.
```

Listing B-1. (Cont'd)

```
  mov bx,offset env_8087 ; CLEAR STATUS WORD FOR ENV RESTORE
  mov byte ptr 2[bx],0
  pop es! pop bp         ; RESTORE 8086 ENVIRONMENT
  pop si! pop di
  pop dx! pop cx
  pop bx! pop ax
  mov ss,ndp_ssreg       ; SWITCH BACK TO PREVIOUS STACK
  mov sp,ndp_spreg
  FLDENV env_8087        ; RESTORE 8087 ENV WITH GOOD STATUS
  FWAIT
  pop ds                 ; RESTORE PREVIOUS DATA SEGMENT
  iret
;
;
in_8087:
;-------
; entry:  DS = SYSDAT
; Only user-specified error conditions generate
; interrupts from the 8087.

  mov bx,owner_8087       ; GET THE PROCESS DESCRIPTOR
  test bx,bx              ; CHECK IF OWNER HAS ALREADY
  jz end_87               ; TERMINATED
  mov si, offset env_8087 ; IF SEVERE ERROR,TERMINATE
  mov ax, statusw[si]     ;
                          ; IF NOT SEVERE,RETURN & CONTINUE
  test ax,03ah            ; 3A = UNDER/OVERFLOW,PRECISION,
  jnz end_87              ;      AND DENORMALIZED OPERAND
  or p_flag[bx],080h      ; NOT 3A = ZERO DIVIDE OR INVALID
                          ; OPERATION (STACK ERROR)
end_87:
  ret
```

End of Appendix B

Index

D

E

F

G

H

T

U

V

W

X

Y

Z