DIGITAL RESEARCH®

Concurrent CP/M™
Operating System
System Guide

Foreword

Concurrent CP/M™ can be configured as a single or multiple user, multitasking, real-time operating system. It is designed for use with any disk-based microcomputer using an Intel® 8086, 8088, or compatible microprocessor with a real-time clock. Concurrent CP/M is modular in design, and can be modified to suit the needs of a particular installation.

Concurrent CP/M also can support many IBM® Personal Computer Disk Operating System (PC DOS) and MS™ -DOS programs. In addition, you can read and write to PC DOS and MS-DOS disks. In this manual, the term DOS refers to both PC DOS and MS-DOS.

The information in this manual is arranged in the order needed for use by the system designer. Section 1 provides an overview of the Concurrent CP/M system. Section 2 describes how to build a Concurrent CP/M system using the GENCCPM utility. Section 3 contains an overview of the Concurrent CP/M Extended Input/Output System (XIOS). XIOS Character Devices are covered in Section 4, and Disk Devices in Section 5. Section 6 describes special character I/O functions needed to support DOS programs.

A detailed description of the XIOS Timer Interrupt routine is found in Section 7. Section 8 deals with debugging the XIOS. Section 9 discusses the bootstrap loader program necessary for loading the operating system from disk. Section 10 treats the utilities that the OEM must write in order to have a commercially distributable system. Section 11 covers changes to end-user documentation which the OEM must make if certain modifications to Concurrent CP/M are performed. Appendix A discusses removable media considerations, and Appendix B covers graphics implementation.

Many sections of this manual refer to the example XIOS. There are two examples provided. One is a single user system to run on the IBM Personal Computer. The other is a multi-user system running on a CompuPro® 86/87 with serial terminals. The single user example includes source code for windowing support for a video mapped display. However windowing is not required for the system. The source code for both examples appears on the Concurrent CP/M distribution disk; we strongly suggest assembling the source files following the instructions in Section 2, and referring often to the assembly listing while reading this manual. Example listings of the Concurrent CP/M Loader BIOS and Boot Sector can also be found on the release disk.

Digital Research® supports the user interface and software interface to Concurrent CP/M, as described in the Concurrent CP/M Operating System User's Guide and the Concurrent CP/M Operating System Programmer's Reference Guide, respectively. Digital Research does not support any additions or modifications made to Concurrent CP/M by the OEM or distributor. The OEM or Concurrent CP/M distributor must also support the hardware interface (XIOS) for a particular hardware environment.

The Concurrent CP/M System Guide is intended for use by system designers who want to modify either the user or hardware interface to Concurrent CP/M. It assumes you have already implemented a CP/M-86® 1.0 Basic Input/Output System (BIOS), preferably on the target Concurrent CP/M machine. It also assumes you are familiar with these four manuals, which document and support Concurrent CP/M:

 ● The Concurrent CP/M Operating System User's Guide documents the user's interface to Concurrent CP/M, explaining the various features used to execute applications programs and Digital Research utility programs.

 ● The Concurrent CP/M Operating System Programmer's Reference Guide documents the applications programmer's interface to Concurrent CP/M, explaining the internal file structure and system entry points—information essential to create applications programs that run in the Concurrent CP/M environment.

 ● The Concurrent CP/M Operating System Programmer's Utilities Guide documents the Digital Research utility programs programmers use to write, debug, and verify applications programs written for the Concurrent CP/M environment.

 ● The Concurrent CP/M Operating System System Guide documents the internal, hardware-dependent structures of Concurrent CP/M.

Standard terminology is used throughout these manuals to refer to Concurrent CP/M features. For example, the names of all XIOS function calls and their associated code routines begin with IO_. Concurrent CP/M system functions available through the logically invariant software interface are called system calls. The names of all data structures internal to the operating system or XIOS are capitalized; for example, XIOS Header and Disk Parameter Block. The Concurrent CP/M system data segment is referred to as the SYSDAT area or simply SYSDAT. The fixed structure at the beginning of the SYSDAT area, documented in Section 1.10 of this manual, is called the SYSDAT DATA.

Table of Contents

Table of Contents
(continued)

Table of Contents
(continued)

Appendixes

Tables, Figures, and Listings

Tables

Tables, Figures, and Listings
(continued)

Figures

Tables, Figures and Listings
(continued)

Listings

Section 1
System Overview

Concurrent CP/M is a multitasking, real-time operating system. It can be configured for one or more user terminals. Each user terminal can run multiple tasks simultaneously on one or more virtual consoles. Concurrent CP/M supports extended features, such as intercommunication and synchronization of independently running processes. It is designed for implementation in a large variety of hardware environments and as such, you can easily customize it to fit a particular hardware environment and/or user's needs.

Concurrent CP/M also supports DOS (PC DOS and MS-DOS) programs and media. The XIOS support for DOS media is described in Section 5 of this manual. DOS character I/O is described in Section 6.

Concurrent CP/M consists of three levels of interface: the user interface, the logically invariant software interface, and the hardware interface. The user interface, which Digital Research distributes, is the Resident System Process (RSP) called the Terminal Message Process (TMP). It accepts commands from the user and either performs those commands that are built into the TMP, or passes the command to the operating system via the Command Line Interpreter (P_CLI). The Command Line Interpreter in the operating system kernel either invokes an RSP or loads a disk file in order to perform the command.

The logically invariant interface to the operating system consists of the system calls as described in the Concurrent CP/M Operating System Programmer's Reference Guide. The logically invariant interface also connects transient and resident processes with the hardware interface.

The physical interface, or XIOS (extended I/O system), communicates directly with the particular hardware environment. It is composed of a set of functions that are called by processes needing physical I/O. Sections 3 through 6 describe these functions. Figure 1-1 shows the relationships among the three interfaces.

Digital Research distributes Concurrent CP/M with machine-readable source code for both the user and example hardware interfaces. You can write a custom user and/or hardware interface, and incorporate them by using the system generation utility, GENCCPM. There are two example XIOSs supplied with the system. One is written for the IBM Personal Computer, as a single user system with multiple virtual consoles. The other XIOS is written for the CompuPro 86/87 with multiple serial terminals. The example XIOSs are designed to be examples and not commercially distributable systems. Wherever a choice between clarity and efficiency is necessary, the examples are written for clarity.

1-1

This section describes the modules comprising a typical Concurrent
CP/M operating system. It is important that you understand this
material before you try to customize the operating system for a
particular application.

```
                              User
                               │
                               ▼
              ┌──────────────────────────────────┐
              │         User Interface           │
              │                                  │
              │            (TMP)                 │
              └──────────────────────────────────┘
                               │
                               ▼
              ┌──────────────────────────────────┐
              │           Invariant              │
              │           Interface              │
              │                                  │
              │     (SUP RTM MEM CIO BDOS)       │
              └──────────────────────────────────┘
                               │
                               ▼
              ┌──────────────────────────────────┐
              │           Hardware               │
              │           Interface              │
              │            (XIOS)                │
              └──────────────────────────────────┘
                               │
                               ▼
                     Hardware Environment
```
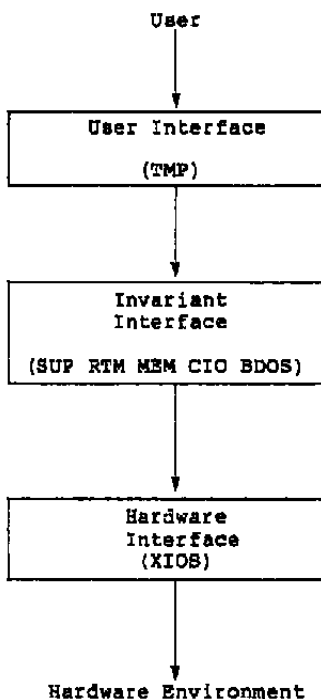
Figure 1-1. Concurrent CP/M Interfacing

1.1   Concurrent CP/M Organization

Concurrent CP/M is composed of six basic code modules. The Real-time Monitor (RTM) handles process-related functions, including dispatching, creation, and termination, as well as the Input/Output system state logic. The Memory module (MEM) manages memory and handles the Memory Allocate (M_ALLOC) and Memory Free (M_FREE) system calls. The Character I/O module (CIO) handles all console and list device functions, and the Basic Disk Operating System (BDOS) manages the file system. These four modules communicate with the Supervisor (SUP) and the Extended Input/Output System (XIOS).

The SUP module manages the interaction between transient processes, such as user programs, and the system modules. All function calls go through a common table-driven interface in SUP. The SUP module also contains the Program Load (P_LOAD) and Command Line Interpreter (P_CLI) system calls.

The XIOS module handles the physical interface to a particular hardware environment. Any of the Concurrent CP/M logical code modules can call the XIOS to perform specific hardware-dependent functions. The names used in this manual for the XIOS functions always begin with IO_ in order to easily distinguish them from Concurrent CP/M operating system calls.

All operating system code modules, including the SUP and XIOS, share a data segment called the System Data Area (SYSDAT). The beginning of SYSDAT is the SYSDAT DATA, a well-defined structure containing public data used by all system code modules. Following this fixed portion are local data areas belonging to specific code modules. The XIOS area is the last of these code module areas. Following the XIOS Area are Table Areas, used for the Process Descriptors, Queue Descriptors, System Flag Tables, and other operating system tables. These tables vary in size depending on options chosen during system generation. See Section 2, "System Generation."

The Resident System Processes (RSPs) occupy the area in memory immediately following the SYSDAT module. The RSPs you select at system generation time become an integral part of the Concurrent CP/M operating system. For more information on RSPs, see Section 1.11 of this manual, and the Concurrent CP/M Operating System Programmer's Reference Guide.

Concurrent CP/M loads all transient programs into the Transient Program Area (TPA). The TPA for a given implementation of Concurrent CP/M is determined at system generation time.

1.2 Memory Layout

The Concurrent CP/M operating system area can exist anywhere in
memory except over the interrupt vector area. You define the exact
location of Concurrent CP/M during system generation. The GENCCPM
program determines the memory locations of the system modules that
make up Concurrent CP/M based upon system generation parameters and
the size of the modules.

The XIOS must reside within SYSDAT. You must write the XIOS as an
8080 model program, with both the code and data segment registers
set to the beginning of SYSDAT.

Figure 1-2 shows the relationship of the Concurrent CP/M system
image to the CCPM.SYS disk file structure.

1.3 Supervisor

The Concurrent CP/M Supervisor (SUP) manages the interface between
system and transient processes and the invariant operating system.
All system calls go through a common table-driven interface in SUP.

The SUP module also contains system calls that invoke other system
calls, like P_LOAD (Program Load) and P_CLI (Command Line
Interpreter).

Table 1-1. Supervisor System Calls

System Call	Number	Hex
F_PARSE	152	98
P_CHAIN	47	2F
P_CLI	150	96
P_LOAD	59	3B
P_RPL	151	97
S_BDOSVER	12	0C
S_BIOS	50	32
B_OSVER	163	0A3
S_SYSDAT	154	9A
S_SERIAL	107	6B
T_SECONDS	155	9B

```
       (top of memory)
   ┌─────────────────────┐
   │∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿│        End of file──→  ┌──────────────────┐
   │∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿│                        │    CCPM.SYS       │
   │         TPA         │                        │   Extra Group     │
   │                     │                        │  (Used to hold    │
   │                     │        End of          │  GENCCPM options) │
   ├─────────────────────┤  ◄──── O.S. Area       └──────────────────┘
   │    Disk Buffers     │
   │                     │        End of O.S. ───→ ┌──────────────────┐
   ├─────────────────────┤  ◄────                  │                  │
   │        RSPs         │                         │                  │
   │                     │                         │   CCPM.SYS       │
   ├─────────────────────┤         ▲               │   Data Group     │
   │     Table Area      │         │               │                  │
   ├─────────────────────┤      within             │                  │
   │        XIOS         │       64k               │                  │
   ├─────────────────────┤         │               ├──────────────────┤
   │    SYSDAT DATA      │         ▼               │                  │
   ├─────────────────────┤  ◄──── XIOS ───→         │                  │
   │      BDOS Code      │      Code & Data         │                  │
   ├─────────────────────┤        Segment           │   CCPM.SYS       │
   │      CIO Code       │                          │   Code Group     │
   ├─────────────────────┤                          │                  │
   │      MEM Code       │                          │                  │
   ├─────────────────────┤                          │                  │
   │      RTM Code       │                          │                  │
   ├─────────────────────┤                          ├──────────────────┤
   │      SUP Code       │                          │                  │
   ├─────────────────────┤  ◄──── beginning ───→     │   CCPM.SYS       │
   │                     │       of O.S. area        │   CMD Format     │
   │         TPA         │                           │   File Header    │
   │∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿│                           └──────────────────┘
   ├─────────────────────┤
   │∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿∿│        0:0400H             (Start of File)
   │  Interrupt Vectors  │
   └─────────────────────┘        0:0000H
```

Figure 1-2. Memory Layout and File Structure

1.4   Real-time Monitor

The Real-time Monitor (RTM) is the multitasking kernel of Concurrent CP/M. It handles process dispatching, queue and flag management, device polling, and system timing tasks. It also manages the logical interrupt system of Concurrent CP/M. The primary function of the RTM is transferring the CPU resource from one process to another, a task accomplished by the RTM dispatcher. At every dispatch operation, the dispatcher stops the currently running process from execution and stores its state in the Process Descriptor (PD) and User Data Area (UDA) associated with that process. The dispatcher then selects the highest-priority process in the ready state and restores it to execution, using the data in its PD and UDA. A process is in the ready state if it is waiting for the CPU resource only. The new process continues to execute until it needs an unavailable resource, a resource needed by another process becomes available, or an external event, such as an interrupt, occurs. At this time the RTM performs another dispatch operation, allowing another process to run.

The Concurrent CP/M RTM dispatcher also performs device polling. A process waits for a polled device through the RTM DEV_POLL system call.

When a process needs to wait for an interrupt, it issues a DEV_WAITFLAG system call on a logical interrupt device. When the appropriate interrupt actually occurs, the XIOS calls the DEV_SETFLAG system call, which wakes up the waiting process. The interrupt routine then performs a Far Jump to the RTM dispatcher, which reschedules the interrupted process, as well as all other ready processes that are not yet on the Ready List. At this point, the dispatcher places the process with the highest priority into execution. Processes that are handling interrupts should run at a better priority than noninterrupt-dependent processes (the lower the priority number, the better the priority) in order to respond quickly to incoming interrupts.

The system clock generates interrupts, clock ticks, typically 60 times per second. This allows Concurrent CP/M to effect process time slicing. Since the operating system waits for the tick flag, the XIOS TICK Interrupt routine must execute a Concurrent CP/M DEV_SETFLAG system call at each tick (see Section 7, "XIOS TICK Interrupt Routine"), then perform a Far Jump to the SUP entry point. At this point, processes with equal priority are scheduled for the CPU resource in round-robin fashion unless a better-priority process is on the Ready List. If no process is ready to use the CPU, Concurrent CP/M remains in the dispatcher until an interrupt occurs, or a polling process is ready to run.

The RTM also handles queue management. System queues are composed
of two parts: the Queue Descriptor, which contains the queue name
and other parameters, and the Queue Buffer, which can contain a
specified number of fixed-length messages. Processes read these
messages from the queue on a first-in, first-out basis. A process
can write to or read from a queue either conditionally or
unconditionally. If a process attempts a conditional read from an
empty queue, or a conditional write to a full one, the RTM returns
an error code to the calling process. However, an unconditional
read or write attempt in these situations causes the suspension of
the process until the operation can be accomplished. The kernel
uses this feature to implement mutual exclusion of processes from
serially reusable system resources, such as the disk hardware.

Other functions of the Real-time Monitor are covered in the
Concurrent CP/M Operating System Programmer's Reference Guide under
their individual descriptions.

Table 1-2.   Real-time Monitor System Calls

System Call	Number	Hex
DEV_SETFLAG	133	85
DEV_WAITFLAG	132	84
DEV_POLL	131	83
P_ABORT	157	9D
P_CREATE	144	90
P_DELAY	141	8D
P_DISPATCH	142	8E
P_PDADR	156	9C
P_PRIORITY	145	91
P_TERM	143	8F
P_TERMCPM	0	00
Q_CREAT	138	8A
Q_CWRITE	140	8C
Q_DELETE	136	88
Q_MAKE	134	86
Q_OPEN	135	87
Q_READ	137	89
Q_WRITE	139	8B

1.5 Memory Management Module

The Memory Management module (MEM) handles all memory functions.
Concurrent CP/M supports an extended model of memory management.
Future releases of Concurrent CP/M might support different versions
of the Memory module depending on classes of memory management
hardware that become available.

The MEM module describes memory partitions internally by Memory
Descriptors (MDs). Concurrent CP/M initially places all available
partitions on the Memory Free List (MFL). Once MEM allocates a
partition (or set of contiguous partitions), it takes that partition
off the MFL and places it on the Memory Allocation List (MAL). The
Memory Allocation List contains descriptions of contiguous areas of
memory known as Memory Allocation Units (MAUs). MAUs always contain
one or more partitions. The MEM module manages the space within an
MAU in the following way: when a process requests extra memory, MEM
first determines if the MAU has enough unused space. If it does,
the extra memory requested comes from the process's own partition
first.

A process can only allocate memory from a MAU in which it already
owns memory, or from a new MAU created from the MFL. If one process
shares memory with another, either can allocate memory from the MAU
that contains the shared memory segment. The MEM module keeps a
count of how many processes "own" a particular memory segment to
ensure that it becomes available within the MAU only when no
processes own it. When all of the memory within an MAU is free, the
MEM module frees the MAU and returns its memory partitions to the
MFL.

If the system for which Concurrent CP/M is being implemented
contains memory management hardware, the XIOS can protect a
process's memory when it is not in context. When the process is
entering the operating system, all memory in the system should be
made Read-Write. When a process is exiting the operating system,
the process's memory should be made Read-Write, the operating system
memory (from CCPMSEG to ENDSEG) made Read-Only, and all other memory
made nonexistent. Memory protection can be implemented within the
XIOS by a routine that intercepts the INT 224 entry point for
Concurrent CP/M system calls, and interrupt routines that handle
attempted memory protection violations.

Figure 1-3 shows how to find a process's memory.

SYSDAT:68H

RLR

PD

MSD

LINK

Next MSD
(0 if none)

(All MSD's pointing to a common
 MAU are grouped together)

MAU   START   LENGTH
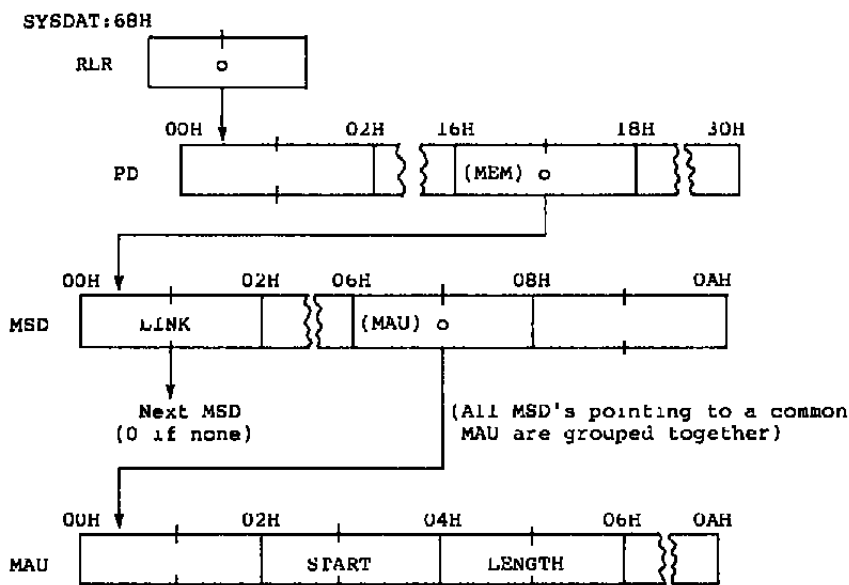
Figure 1-3.   Finding a Process's Memory

Table 1-3. Definitions for Figure 1-3.

Data Field	Explanation
RLR	Ready List Root; points to currently running process.
PD	Process Descriptor; describes a process.
MEM	MEM field of Process Descriptor.
MSD	Memory Segment Descriptor; describes a single memory allocation. A process may have many of these in a linked list. The MSD list pointed to by the MEM field describes all the successful memory allocations made by the process. Also, many MSDs may point to the same MAU. All MSDs pointing to the same MAU are grouped together.
MAU	Memory Allocation Unit; describes a contiguous area of allocated memory. A MAU is built from one or more contiguous memory partitions. The START and LENGTH fields are the starting paragraph and number of paragraphs, respectively.

Table 1-4. Memory Management System Calls

System Call	Number	Hex
M_ALLOC	128, 129	80, 81
M_FREE	130	82
MC_ABS	54	36
MC_ALLFREE	58	3A
MC_ALLOC	55	37
MC_ALLOCABS	56	38
MC_FREE	57	39
MC_MAX	53	35

Note: The MC_ABS, MC_ALLOC, MC_ALLOCABS, MC_FREE, MC_ALLFREE, and MC_MAX system calls internally execute the M_ALLOC and M_FREE system calls. They are supported for compatibility with the CP/M-86 and MP/M-86™ operating systems.

1.6 Character I/O Manager

The Character Input/Output (CIO) module of Concurrent CP/M handles all console and list device I/O, and interfaces to the XIOS, the PIN (Physical Input Process) and the VOUT (Virtual OUTput process). There is one PIN for each user terminal, and one VOUT for each virtual console in the system. An overview of the CIO is presented in the Concurrent CP/M Operating System Programmer's Reference Guide, and XIOS Character Devices are described in Section 4 of this manual. For details of the Console Control Block (CCB) and List Control Block (LCB) data structures, see Sections 4.1 and 4.3 respectively.

Table 1-5. Character I/O System Calls

System Call	Number	Hex
C_ASSIGN	149	95
C_ATTACH	146	92
C_CATTACH	162	0A2
C_DELIMIT	110	6E
C_DETACH	147	93
C_GET	153	99
C_MODE	109	6D
C_RAWIO	6	06
C_READ	1	01
C_READSTR	10	0A
C_SET	148	94
C_STAT	11	0B
C_WRITE	2	02
C_WRITEBLK	111	6F
C_WRITESTR	9	09
L_ATTACH	158	9E
L_CATTACH	161	0A1
L_DETACH	159	9F
L_GET	164	0A4
L_SET	160	0A0
L_WRITE	5	05
L_WRITEBLK	112	70

1.7 Basic Disk Operating System

The Basic Disk Operating System (BDOS) handles all file system functions. It is described in detail in the Concurrent CP/M Operating System Programmer's Reference Guide. Table 1-6 lists the Concurrent CP/M BDOS system calls.

Table 1-6. BDOS System Calls

System Call	Number	Hex
DRV_ACCESS	38	26
DRV_ALLOCVEC	27	1B
DRV_DPB	31	1F
DRV_FLUSH	48	30
DRV_GET	25	19
DRV_GETLABEL	101	65
DRV_LOGINVEC	24	18
DRV_RESET	37	25
DRV_ROVEC	29	1D
DRV_SET	14	0E
DRV_SETLABEL	100	64
DRV_SETRO	28	1E
DRV_SPACE	46	2E
F_ATTRIB	30	1E
F_CLOSE	16	10
F_DELETE	19	13
F_DMASEG	51	33
F_DMAGET	52	34
F_DMAOFF	26	1A
F_ERRMODE	45	2D
F_LOCK	42	2A
F_MAKE	22	16
F_MULTISEC	44	2C
F_OPEN	15	0F
F_PASSWD	106	6A
F_READ	20	14
F_READRAND	33	21
F_RANDREC	36	24
F_RENAME	23	17
F_SFIRST	17	11
F_SIZE	35	23
F_SNEXT	18	12
F_TIMEDATE	102	66
F_TRUNCATE	99	63
F_UNLOCK	43	2B
F_USERNUM	32	20
F_WRITE	21	15
F_WRITERAND	34	22
F_WRITEXFCB	103	67
F_WRITEZF	40	28
T_GET	105	69
T_SET	104	68

1.8   Extended I/O System

The Extended Input/Output System (XIOS) handles the physical
interface to Concurrent CP/M. It is similar to the CP/M-86 BIOS
module, but it is extended in several ways. By modifying the XIOS,
you can run Concurrent CP/M in a large variety of different hardware
environments. The XIOS recognizes two basic types of I/O devices:
character devices and disk drives. Character devices are devices
that handle one character at a time, while disk devices handle
random blocked I/O using data blocks sized from one physical disk
sector to the number of physical sectors in 16K bytes. Use of
devices that vary from these two models must be implemented within
the XIOS. In this way, they appear to be standard Concurrent CP/M
I/O devices to other operating system modules through the XIOS
interface. Sections 4 through 6 contain detailed descriptions of
the XIOS functions, and the source code for two sample
implementations can be found in machine-readable format on the
Concurrent CP/M OEM release disk.

1.9   Reentrancy in the XIOS

Concurrent CP/M allows multiple processes to use certain XIOS
functions simultaneously. The system guarantees that only one
process uses a particular physical device at any given time.
However, some XIOS functions handle more than one physical device,
and thus their interfaces must be reentrant. An example of this is
the IO_CONOUT Function. The calling process passes the virtual
console number to this function. There can be several processes
using the function, each writing a character to a different virtual
console or character device. However, only one process is actually
outputting a character to a given device at any time.

IO_STATLINE can be called more than once. The CLOCK process calls
the IO_STATLINE function once per second, and the PIN process will
also call it on screen switches, CTRL-S, CTRL-P, and CTRL-O.

Since the XIOS file functions, IO_SELDSK, IO_READ, IO_WRITE, and
IO_FLUSH are protected by the MXdisk mutual exclusion queue, only
one process may access them at a time. None of these XIOS
functions, therefore, need to be reentrant.

1.10 SYSDAT Segment

The System Data Area (SYSDAT) is the data segment for all modules of
Concurrent CP/M.   The SYSDAT segment is composed of three main
areas, as shown in Figure 1-4. The first part is the fixed-format
portion, containing global data used by all modules. This is the
SYSDAT DATA. It contains system variables, including values set by
GENCCPM and pointers to the various system tables. The Internal
Data portion contains fields of data belonging to individual
operating system modules. The XIOS begins at the end of this second
area of SYSDAT. The third portion of SYSDAT is the System Table
Area, which is generated and initialized by the GENCCPM system
generation utility. | :

Figure 1-4 shows the relationships among the various parts of
SYSDAT.

```
                        +----------------------+
                        |                      |
                        |     Table Area       |
                        |                      |
                        +----------------------+
                        |                      |
                        |       XIOS           |
                        |                      |
            C00H:       +----------------------+
                        |                      |
                        |   Internal Data      |
                        |                      |
            0B0H:       +----------------------+
                        |                      |
                        |   (SYSDAT DATA)      |
                        |                      |
            000H:       +----------------------+
```
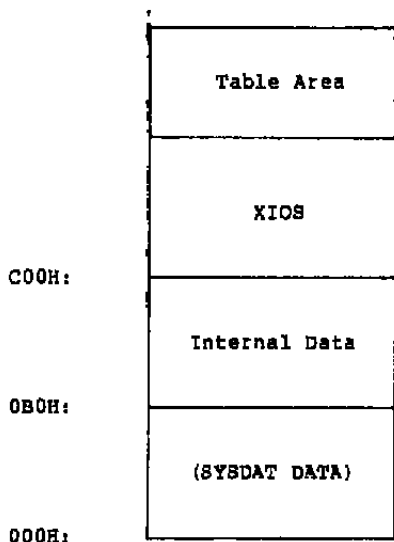
Figure 1-4.   SYSDAT

Figure 1-5 gives the format of the SYSDAT DATA and describes its
data fields.

00H	SUP ENTRY			RESERVED			
08H	RESERVED						
10H	RESERVED						
18H	RESERVED						
20H	RESERVED						
28H	XIOS ENTRY			XIOS INIT			
30H	RESERVED						
38H	DISPATCHER			PDISP			
40H	CCPMSEG		RSPSEG		ENDSEG		RESERVED	NVCNS
48H	NLCB	NCCB	N_FLAGS	SYS_DISK	MMP		RESERVED	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		RESERVED	
90H	RESERVED							
98H	RESERVED							XPCNS
A0H	OFF_8087		SEG_8087		SYS_87_OF		SYS_87_SG	

Figure 1-5. SYSDAT DATA

Table 1-7. SYSDAT DATA Data Fields

Data Field	Explanation
SUP ENTRY	Double-word address of the Supervisor entry point for intermodule communication. All internal system calls go through this entry point.
XIOS ENTRY	Double-word address of the Extended I/O System entry point for intermodule communication. All XIOS function calls go through this entry point.
XIOS INIT	Double-word address of the Extended I/O System Initialization entry point. System hardware initialization takes place 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 (Interrupt Return) instruction. The Dispatcher routine causes a dispatch to occur and then executes an Interrupt Return. All registers are preserved and one level of stack is used. The address in this location can be used by XIOS interrupt handlers for termination instead of executing an IRET instruction. The TICK interrupt handler (I_TICK in the example XIOS's) ends with a Jump Far (JMPF) to the address in this location. Usually, interrupt handlers that make DEV_SETFLAG calls end with a jump far to the address stored in the DISPATCHER field. Refer to the example XIOS interrupt routines and Sections 3.5 and 3.6 for more detailed information.
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 might be wanted by a waiting process.

Table 1-7.   (continued)

Data Field	Explanation
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 the system has been initialized, this field is zero. See the Concurrent CP/M Operating System Programmer's Reference Guide section on debugging RSPs for more information.
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.
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 by GENCCPM.
SYSDISK	Default system disk. The CLI (Command Line Interpreter) looks on this disk if it cannot open the command file on the user's current default disk. Set by GENCCPM.
MMP	Maximum memory allowed per process. Set during GENCCPM.
DAY FILE	Day File option. If this field is 0FFH, the operating system displays date and time information when an RSP or CMD file is invoked. Set by GENCCPM.

Table 1-7. (continued)

Data Field	Explanation
TEMP DISK	Default temporary disk. Programs that create temporary files should use this disk. Set by GENCCPM.
TICKS/SEC	The number of system ticks per second.
LUL	Locked Unused List. Link 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. Link list root of unused Memory Descriptors.
MFL	Memory Free List. Link list root of free memory partitions.
PUL	Process Unused List. Link list root of unused Process Descriptors.
QUL	Queue Unused List. Link list root of unused Queue Descriptors.
QMAU	Queue buffer Memory Allocation Unit.
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.

Table 1-7.   (continued)

Data Field	Explanation
QLR	Queue List Root. Linked list of all System QDs.
MAL	Memory Allocation List; link 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 CP/M version number (returned by the S_BDOSVER system call).
CCPMVERNUM	Concurrent CP/M version number (system call 163, 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 that have open files on them. Each bit of the word value represents a disk drive; the least significant bit represents Drive A, and so on through the most significant bit, Drive P. Bits which are set indicate drives containing open files.

Table 1-7. (continued)

Data Field	Explanation
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	Process currently owning the 8087. Set to 0 if 8087 is not owned. Set to 0FFFFH if no 8087 present.
XPCNS	Number of physical consoles.
OFF_8087	Offset of the 8087 interrupt vector in low memory.
SEG_8087	Segment of the 8087 interrupt vector in low memory.
SYS_87_OF	Offset of the default 8087 exception handler.
SYS_87_SG	Segment of the default 8087 exception handler.

1.11 Resident System Processes

Resident System Processes (RSPs) are an integral part of the Concurrent CP/M operating system. At system generation, the GENCCPM RSP List menu lets you select which RSPs to include in the operating system. GENCCPM then places all selected RSPs in a contiguous area of RAM starting at the end of SYSDAT. The main advantage of an RSP is that it is permanently resident within the Operating System Area, and does not have to be loaded from disk whenever it is needed.

Concurrent CP/M automatically allocates a Process Descriptor (PD) and User Data Area (UDA) for a transient program, but each RSP is responsible for the allocation and initialization of its own PD and UDA. Concurrent CP/M uses the PD and QD structures declared within an RSP directly if they fall within 64K of the SYSDAT segment address. If outside 64K, the RSP's PD and QD are copied to a PD or QD allocated from the Process Unused List or the Queue Unused List. In either case the PD and QD of the RSP lie within 64K of the beginning of the SYSDAT Segment. This allows RSPs to occupy more area than remains in the 64K SYSDAT segment.

Further details on the creation and use of RSPs can be found in the
Concurrent CP/M Operating System Programmer's Reference Guide.

End of Section 1

Section 2
System Generation

The Concurrent CP/M XIOS should be written as an 8080 model (mixed code and data) program and origined at location 0C00H using the ASM86 ORG assembler directive. Once you have written or modified the XIOS source for a particular hardware configuration, use the Digital Research assembler ASM-86™ or RASM-86™ to generate an XIOS.CON file for use with GENCCPM:

    A>ASM86 XIOS   ; Assemble the XIOS

    A>GENCMD XIOS 8080   ; Create XIOS.CMD from XIOS.H86

    A>REN XIOS.CON=XIOS.CMD ; Rename XIOS.CMD to XIOS.CON

Then invoke the GENCCPM program to produce a system image in the CCPM.SYS file by typing the command:

    A>GENCCPM   ; generate system image

2.1 GENCCPM Operation

You can generate a Concurrent CP/M system by running the GENCCPM program under an existing CP/M or Concurrent CP/M system. GENCCPM builds the CCPM.SYS file, which is an image of the Concurrent CP/M operating system. Then you can use DDT-86™ or SID-86™ to place the CCPM.SYS file in memory for debugging under CP/M-86.

GENCCPM allows the user to define certain hardware-dependent variables, the amount of memory to reserve for system data structures, the selection and inclusion of Resident System Processes in the CCPM.SYS file, and other system parameters. The first action GENCCPM performs is to check the current default drive for the files necessary to construct the operating system image:

- SUP.CON   Supervisor Code Module
- RTM.CON   Real Time Monitor Code Module
- MEM.CON   Memory Manager Code Module
- CIO.CON   Character Input/Output Code Module
- BDOS.CON   Basic Disk Operating System Code Module
- XIOS.CON   Extended Input/Output System Module
- SYSDAT.CON   SYSDAT DATA and Internal Data modules of
                SYSDAT segment

- VOUT.RSP   Virtual console OUTput process
- PIN.RSP   Physical keyboard INput process
- TMP.RSP   Terminal Message Process
- CLOCK.RSP   CLOCK process
- DIR.RSP   DIRectory process
- ABORT.RSP   ABORT process

Note: *.RSP = Resident System Process file. The VOUT, PIN, TMP,
and CLOCK RSPs are required for Concurrent CP/M to run. The RSPs
listed are all distributed with Concurrent CP/M.

If GENCCPM does not find the preceding .CON files on the default
drive, it prints an error message on the console:

      Can't find these modules: <FILESPEC>...{<FILESPEC>}

where FILESPEC is the name of the missing file.

2.2 GENCCPM Main Menu

All of the GENCCPM Main Menu options have default values. When
generating a system, GENCCPM assumes the value shown in square
brackets, unless you specify another value. Any menu item that
requires a yes or no response represents a Boolean value, and can be
toggled simply by entering the variable. For example, entering
VERBOSE in response to the GENCCPM prompt will change the state of
the VERBOSE variable from the default state, [Y], to the opposite
state.

In the GENCCPM Main Menu illustrated in Figure 2-1, all numeric
values are in hexadecimal notation.

      *** Concurrent CP/M 3.1 GENCCPM Main Menu ***

          help   GENCCPM Help
       verbose [Y]   More Verbose GENCCPM Messages
     destdrive [A:]   CCPM.SYS Output To (Destination) Drive
     deletesys [N]   Delete (instead of rename) old CCPM.SYS file

       sysparams   Display/Change System Parameters
          memory   Display/Change Memory Allocation Partitions
     diskbuffers   Display/Change Disk Buffer Allocation
         oslabel   Display/Change Operating System Label
            rsps   Display/Change RSP List

          gensys   I'm finished changing things, go GEN a SYStem

    Changes?__

                  Figure 2-1. GENCCPM Main Menu

If you type HELP in response to the GENCCPM Main Menu prompt Changes?, as shown in this example:

    Changes? **HELP** <cr>

the program prints the following message on the Help Function Screen:

                    *** GENCCPM Help Function ***
                    ============================

    GENCCPM lets you edit and generate a system image from
    operating system modules on the default disk drive.   A
    detailed explanation of each GENCCPM parameter may be
    found in the Concurrent CP/M System Guide, Section 2.

    GENCCPM assumes the default values shown within square
    brackets. All numbers are in Hexadecimal. To change a
    parameter, enter the parameter name followed by "=" and
    the new value. Type <cr> (carriage return) to enter the
    assignment. You can make multiple assignments if you
    separate them by a space. No spaces are allowed within
    an assignment. Example:

    Changes? verbose=N sysdrive=A: openmax=1A <cr>

    Parameter names may be shortened to the minimum
    combination of letters unique to the currently displayed
    menu. Example:

    Changes? v=N des=A: del=Y <cr>

    Press RETURN to continue...__

            Figure 2-2. GENCCPM Help Function Screen 1

Sub-menus (the last few options) are accessed by typing
the sub-menu name followed by <cr>.   You may enter
multiple sub- menus, in which case each sub-menu will be
displayed in order.   Example:

Changes? help sysparams reps <cr>

Enter <cr> alone to exit a menu, or a parameter name, "="
and the new value to assign a parameter.   Multiple
assignments may be entered, as in response to the Main
Menu prompt.

Press RETURN to continue.__

Figure 2-3.   GENCCPM Help Function Screen 2

Table 2-1 describes the remaining GENCCPM Main Menu options.

Table 2-1.   GENCCPM Main Menu Options

Option	Explanation
VERBOSE	The GENCCPM program messages are normally verbose. However, experienced operators might want to limit them in the interest of efficiency. Setting VERBOSE to N (no) limits the length of GENCCPM messages to the absolute minimum.
DESTDRIVE	The drive upon which the generated CCPM.SYS file is to reside. If no destination drive is specified, GENCCPM assumes the currently logged drive as the default.
DELETESYS	Delete, instead of rename, old CCPM.SYS file. Normally, GENCCPM renames the previous system file to CCPM.OLD before building the new system image. By specifying DELETESYS=Y, you cause GENCCPM to delete the old file instead. This is useful when disk space is limited.
SYSPARAMS	Typing SYSPARAMS <cr> displays the GENCCPM System Parameter Menu. See Figure 2-4 and accompanying text.

Table 2-1.   (continued)

Option	Explanation
MEMORY	Typing MEMORY <cr> displays the GENCCPM Memory Partition Menu. See Figure 2-5 and accompanying text.
DISKBUFFERS	Typing DISKBUFFERS <cr> displays the GENCCPM Disk Buffer Allocation Menu. See Figure 2-7 and accompanying text.
OSLABEL	Typing OSLABEL <cr> displays the GENCCPM Operating System Label Menu. See Figure 2-8 and accompanying text.
RSPS	Typing RSPS <cr> displays the GENCCPM RSP List Menu. See Figure 2-6 and accompanying text.
GENSYS	Typing GENSYS <cr> initiates the GENeration of the SYStem file. When using an input file to specify system parameters, and the GENSYS command is not the last line in the input file, GENCCPM goes into interactive mode and prompts you for any additional changes. See Section 2.9, "GENCCPM Input Files," for more information.

Note: To create the CCPM.SYS file you must type in the GENSYS command, or include it in the GENCCPM input file.

2.3   System Parameters Menu

The GENCMD System Parameters Menu is shown in Figure 2-3.   You access this menu by typing SYSPARAMS in response to the Main Menu.

Note: All GENCCPM parameter values are in hexadecimal.

Display/Change System Parameters Menu

```
   sysdrive [B:]   System Drive
   tmpdrive [B:]   Temporary File Drive
 cmdlogging [N]    Command Day/File Logging at Console
 compatmode [Y]    CP/M FCB Compatibility Mode
    memmax [4000]  Maximum Memory per Process (paragraphs)
   openmax [20]    Open Files per Process Maximum
   lockmax [20]    Locked Records per Process Maximum

   osstart [1008]  Starting Paragraph of Operating System
 nopenfiles [  40] Number of Open File and Locked Record Entries
   npdescs [14]    Number of Process Descriptors
      nqcbs [20]   Number of Queue Control Blocks
  qbufsize [ 400]  Queue Buffer Total Size in bytes
    nflags [20]    Number of System Flags
Changes?__
```

Figure 2-4.   GENCCPM System Parameters Menu.

Table 2-2.   System Parameters Menu Options

Option	Explanation
SYSDRIVE	The system drive where Concurrent CP/M looks for a transient program when it is not found on the current default drive. All the commonly used transient processes can thus be placed on one disk under User Number 0 and are not needed on every drive and user number. See the Concurrent CP/M Operating System User's Guide for information on how the operating system performs file searches.
TMPDRIVE	The drive entered here is used as the drive for temporary disk files. This entry can be accessed in the System Data Segment by application programs as the drive on which to create temporary files. The temporary drive should be the fastest drive in the system, for example, the Memory Disk, if implemented.

Table 2-2.   (continued)

Option	Explanation
CMDLOGGING	Entering the response [Y] causes the generated Concurrent CP/M Command Line Interpreter (CLI) to display the current time and how the command will be executed.
COMPATMODE	CP/M® FCB Compatibility Mode [Y]. When the default value [Y] is set, the operating system recognizes the compatibility attributes. Setting this parameter to [N] makes the generated system ignore the compatibility attributes. See the Concurrent CP/M Operating System Programmer's Reference Guide, Section 2.12, "Compatibility Attributes," for more information on this feature.
MEMMAX	Maximum Paragraphs Per Process [4000]. A process may make Concurrent CP/M memory allocations. This parameter puts an upper limit on how much memory any one process can obtain. The default shown here is 256K (40000H) bytes.
OPENMAX	Maximum Open Files per Process [20]. This parameter specifies the maximum number of files that a single process, usually one program, can open at any given time. This number can range from 0 to 255 (0FFH) and must be less than or equal to the total open files and locked records for the system. See the explanation of the NOPENFILES parameter below.
LOCKMAX	Maximum Locked Records per Process [20]. This parameter specifies the maximum number of records that a single process, usually one program, can lock at any given time. This number can range from 0 to 255 (0FFH) and must be less than or equal to the total open files and locked records for the system. See the explanation of the NOPENFILES parameter in the SYSPARAMS Menu.

Table 2-2. (continued)

Option	Explanation
OSSTART	Starting Paragraph of the operating system [100B]. The starting paragraph is where the CCPMLDR is to put the operating system. Code execution starts here, with the CS register set to this value and the IP register set to 0. The Data Segment Register is set to the SYSDAT segment address. When first bringing up and debugging Concurrent CP/M under CP/M-86, the answer to this question should be 8 plus where DDT-86 running under CP/M-86 reads in the file using the R command. The DDT86 R command also can be used to read the CCPM.SYS file to a specific memory location. After debugging the system, you might want to relocate it to an address more appropriate to your hardware configuration. This location naturally depends on where the Boot Sector and Loader are placed, and how much RAM is used by ROM monitor or memory-mapped I/O devices.
NOPENFILES	Total Open Files in System [40]. This parameter specifies the total size of the System Lock List, which includes the total number of open disk files plus the total number of locked records for all the processes executing under Concurrent CP/M at any given time. This number must be greater than or equal to the maximum open files per process (the OPENMAX parameter above) and the maximum locked records per process (the LOCKMAX parameter above). It is possible either to allow each process to use up the total System Lock List space, or to allow each process to only open a fraction of the system total. The first technique implies a situation where one process can forcibly block others because it has consumed all the available Lock list items.

Table 2-2. (continued)

Option	Explanation
NPDESCS	Number Of Process Descriptors [14]. For each memory partition, at least one transient program can be loaded and run. If transient programs create child processes, or if RSPs extend past 64K from the beginning of SYSDAT, extra Process Descriptors are needed. When first bringing up and debugging Concurrent CP/M, the default for this parameter suffices. After the debug phase, during system tuning, you can use the Concurrent CP/M SYSTAT Utility to monitor the number of processes and queues in use by the system at any time.
NQCBS	Number Of Queue Control Blocks [20]. The number of Queue Control Blocks should be the maximum number of queues that may be created by transient programs or RSPs outside of 64k from SYSDAT. The default value suffices during initial system debugging.
QBUFSIZE	Size Of Queue Buffer Area in Bytes [400]. The Queue Buffer Area is space reserved for Queue Buffers. The size of the buffer area required for a particular queue is the message length times the number of messages. The Queue Buffer Area should be the anticipated maximum that transient programs will need. Again, the default value will be adequate for initial system debugging. Note that the Queue Buffer Area can be large enough (up to 0FFFFH) to extend past the SYSDAT 64K boundary.
NFLAGS	Size of the flag table [20]. Flags are three-byte semaphores used by interrupt routines. The number of flags needed depends on the design of the XIOS. More information on using flags for interrupt devices can be found in Section 3 under "Interrupt Devices". See also the Concurrent CP/M Operating System Programmer's Guide on Dev_flagset, Dev_flagwt.

2.4   Memory Allocation Menu

The Memory Allocation Partitions Menu, shown in Figure 2-5, is an
interactive menu. When the menu is first displayed, it lists the
current memory partitions. If none have been specified, the list
field is blank. Following the list is the menu of options
available. You may choose either to ADD to the list of partitions,
or to DELETE one or more partitions. Partition assignments must be
made by specifying either ADD or DELETE, followed by an equal sign,
the starting address and last address of the memory region to be
partitioned, and the size, in paragraphs, of each partition. All
values must be in hexadecimal notation and separated by commas. An
asterisk can be used to delete all memory partitions. The Start and
Last values are paragraph addresses; multiply them by 16 (10H) to
obtain absolute addresses. Similarly, partition sizes are in
paragraphs; multiply by 16 (10H) to obtain size in bytes.

In the example below, all default memory partitions are first
deleted (DELETE=*). Then two kinds of memory partitions are added
to the list: 16K (4000h) partitions from address 2400:0 to 4000:0,
and 32K (8000h) partitions from 4000:0 to 6000:0.

```
                 Addresses            Partitions (in paragraphs)
     #      Start     Last        Size      Qty
     1.     400h      6000h       400h      17h


     Display/Change Memory Allocation Partitions
          add             ADD memory partition(s)
          delete          DELETE memory partition(s)

     Changes? delete=* add=2400,4000,400 add=4000,6000,800

                 Addresses            Partitions
     #      Start     Last        Size      Qty
     1.     2400h     4000h       400h      7h
     2.     4000h     6000h       800h      4h


     Display/Change Memory Allocation Partitions
          add             ADD memory partition(s)
          delete          DELETE memory partition(s)

     Changes? <cr>
```

Figure 2-5. GENCCPM Memory Allocation Sample Session

Memory partitions are highly dependent on the particular hardware
environment. Therefore, you should carefully examine the defaults
that are given, and change them if they are inappropriate. The
memory partitions cannot overlap, nor can they overlap the operating
system area. GENCCPM checks and trims memory partitions that
overlap the operating system but does not check for partitions that
refer to nonexistent system memory. GENCCPM does not size existing
memory because the hardware on which it is running might be
different from the target Concurrent CP/M machine (this might be
done by the XIOS at initialization time). Error messages are
displayed in case of overlapping or incorrectly sized partitions,
but GENCCPM does not automatically trim overlapping memory
partitions. GENCCPM does not allow you to exit the Main Menu or the
Memory Allocation Menu if the memory partition list is not valid.

The nature of your application dictates how you should specify the
partition boundaries in your system. The system never divides a
single partition among unrelated programs. If any given memory
request requires a memory segment that is larger than the available
partitions, the system concatenates adjoining partitions to form a
single contiguous area of memory. The MEM module algorithm that
determines the best fit for a given memory allocation request takes
into account the number of partitions that will be used and the
amount of unused space that will be left in the memory region. This
allows you to evaluate the tradeoffs between memory allocation
boundary conditions causing internal versus external memory
fragmentation, as described below.

External memory fragmentation occurs when memory is allocated in
small amounts. This can lead to a situation where there is plenty
of memory but no contiguous area large enough to load a large
program. Internal fragmentation occurs when memory is divided into
large partitions, and loading a small program leaves large amounts
of unused memory in the partition. In this case, a large program
can always load if a partition is available, but the unused areas
within the large partitions cannot be used to load small programs if
all partitions are allocated.

When running GENCCPM you can specify a few large partitions, many
small partitions, or any combination of the two. If a particular
environment requires running many small programs frequently and
large programs only occasionally, memory should be divided into
small partitions. This simulates dynamic memory management as the
partitions become smaller. Large programs are able to load as long
as memory has not become too fragmented. If the environment
consists of running mostly large programs or if the programs are run
serially, the large-partition model should be used. The choice is
not trivial and might require some experimentation before a
satisfactory compromise is attained. Typical solutions divide
memory into 4K to 16K partitions.

2.5   GENCCPM RSP List Menu

The GENCCPM RSP (Resident System Process) List Menu is shown in
Figure 2-6. The example session illustrates excluding ABORT.RSP and
MY.RSP from the list of RSPs to be included in the system.

RSPs to be included are:

        PIN.RSP   DIR.RSP   ABORT.RSP   TMP.RSP
        VOUT.RSP   CLOCK.RSP   MY.RSP

Display/Change RSP List

     include   Include RSPs
     exclude   Exclude RSPs

Changes?__exclude=abort.rsp,my.rsp

RSPs to be included are:

        PIN.RSP   DIR.RSP   VOUT.RSP   CLOCK.RSP
        TMP.RSP

Changes?__<cr>

Figure 2-6.   GENCCPM RSP List Menu Sample Session

The GENCCPM RSP List Menu first reads the directory of the current
default disk and lists all .RSP files present. Responding to the
GENCCPM prompt Changes? with either an include or exclude command
edits the list of RSPs to be made part of the operating system at
system generation time. The wildcard (*:) file specification can be
used with the include command to automatically include all .RSP
files on the disk.

Note: The PIN, VOUT, and CLOCK RSPs must be included for Concurrent
CP/M to run.

2.6 GENCCPM OSLABEL Menu

If you type OSLABEL in response to the main menu prompt, as shown in
this example:

    Changes? **OSLABEL**

the following screen menu appears on your screen:

    Display/Change Operating System Label
    Current message is:
    <null>

    Add lines to message. Terminate by entering only RETURN:

    Figure 2-7. GENCCPM Operating System Label Menu

You can type any message at this point. This message is printed on
each virtual console when the system boots up. Note that if the
message contains a $, GENCCPM accepts it, but it causes the
operating system to terminate the message when it is being printed.
This is because the operating system uses the C_WRITESTR function to
print the message, and $ is the default message terminator.

The XIOS might also print its own sign-on message during the INIT
routine. In this case, the XIOS message appears before the message
specified in the GENCCPM OSLABEL Menu.

2.7 GENCCPM Disk Buffering Menu

Typing DISKBUFFERS in response to the main menu prompt displays the
GENCCPM Disk Buffering Menu. Figure 2-8 shows a sample session:

```
              *** Disk Buffering Information ***
            Dir  Max/Proc   Data Max/Proc   Hash   Specified
     Drv    Bufs Dir Bufs   Bufs Dat Bufs   -ing   Buf Pgphs
     ===    ==== ========   ==== ========   ====   =========
     A:      ??     0        ??     0        yes      ??
     B:      ??     0        ??     0        yes      ??
     C:      ??     0        ??     0        yes      ??
     D:      ??     0        ??     0        yes      ??
     E:      ??     0        ??     0        yes      ??
     M:      ??     0       fixed           fixed     ??
              Total paragraphs allocated to buffers: 0
     Drive (<cr> to exit) ? a:
     Number of directory buffers, or drive to share with? 8
     Maximum directory buffers per process [8] ? 4
     Number of data buffers, or drive to share with? 4
     Maximum data buffers per process [4]? 2
     Hashing [yes] ? <cr>

              *** Disk Buffering Information ***
            Dir  Max/Proc   Data Max/Proc   Hash   Specified
     Drv    Bufs Dir Bufs   Bufs Dat Bufs   -ing   Buf Pgphs
     ===    ==== ========   ==== ========   ====   =========
     A:      8      4         4     2        yes     200
     B:      ??     0        ??     0        yes      ??
     C:      ??     0        ??     0        yes      ??
     D:      ??     0        ??     0        yes      ??
     E:      ??     0        ??     0        yes      ??
     M:      ??     0       fixed           fixed     ??
    Total paragraphs allocated to buffers: 200
     Drive (<cr> to exit) ? *:
     Number of directory buffers, or drive to share with? a:
     Number of data buffers, or drive to share with? a:
     Hashing [yes] ? <cr>

              *** Disk Buffering Information ***
            Dir  Max/Proc   Data Max/Proc   Hash   Specified
     Drv    Bufs Dir Bufs   Bufs Dat Bufs   -ing   Buf Pgphs
     ===    ==== ========   ==== ========   ====   =========
     A:      8      4         4     2        yes     200
     B:     shares A:        shares A:       yes      80
     C:     shares A:        shares A:       yes      20
     D:     shares A:        shares A:       yes      18
     E:     shares A:        shares A:       yes      10
     M:     shares A:       fixed           fixed      0
    Total paragraphs allocated to buffers: 2C8

     Drive (<cr> to exit) ? <cr>
```

Figure 2-8. GENCCPM Disk Buffering Sample Session

In the sample session shown in Figure 2-8, GENCCPM is reading the
DPH addresses from the XIOS Header, and calculating the buffer
parameters based upon the data in the DPHs and the answers to its
questions. GENCCPM only asks questions for the relevant fields in
the DPH that you have marked with 0FFFFh values. See Section 5.4,
"Disk Parameter Header," for a detailed explanation of DPH fields
and GENCCPM table generation. An asterisk can be used to specify
all drives, in which case GENCCPM applies your answers to the
following questions to all unconfigured drives.

Note that GENCCPM prints out how many bytes of memory must be
allocated to implement your disk buffering requests. You should be
aware that disk buffering decisions can significantly impact the
performance and efficiency of the system being generated.   If
minimizing the amount of memory occupied by the system is an
important consideration, you can use the Disk Buffering Menu to
specify a minimal disk buffer space. We have found, however, that
the amount of Directory Hashing space allocated has the most impact
on system performance, followed by the amount of Directory Buffer
space allocated.   As with the trade-offs in memory partition
allocation discussed above, deciding on the proper ratio of
operating system space to performance requires some experimentation.

Note also that if DOS media is supported, directory hashing space
must be allocated for the DOS file allocation table (FAT).   See
Section 5.5.1 for information on allocating enough space for the FAT
and the hash table.

GENCCPM checks to see that the relevant fields in the DPHs are no
longer set to 0FFFFH. GENCCPM does not allow you to exit from the
Main Menu until these fields have been set using the Disk Buffering
Menu.

2.8   GENCCPM GENSYS Option

Finally, specifying the GENSYS option in answer to the main menu
prompt causes GENCCPM to generate the system image on the specified
destination disk drive. During the actual system generation, the
following messages print out on the screen:

```
        Generating new SYS file
        Generating tables
        Appending RSPs to system file
        Doing Fixups
        SYS image load map:
                Code starts at GGGGh
                Data starts at HHHHh
              Tables start at IIIIh
                RSPs start at JJJJh
        XIOS Buffers start at KKKKh
                End of OS at LLLLh
```

Trimming memory partitions. New List:

	Addresses		Partitions		
	(in Paragraphs)		Size	How	(only if
#	Start	Last	(Params.)	Many	necessary)
1.	AAAAh	BBBBh	XXXXh	Yh	
2.	MMMMh	NNNNh	QQQQh	Vh	

Wrapping up

A>
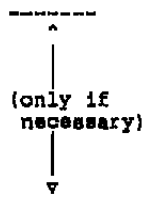
2.9   GENCCPM Input Files

GENCCPM allows you to input all system generation commands from an
input file. You can also redirect the console output to a disk
file. You use these GENCCPM features by invoking it with command of
the form:

        GENCCPM <filein >fileout

where filein is the name of the GENCCPM input file. Note that no
spaces can intervene between the greater-than or less-than sign and
the file specification. If this condition is not met, GENCCPM
responds with the message:

        REDIRECTION ERROR

The format of the input file is similar to a SUBMIT file; each
command is entered on a separate line, followed by a carriage
return, exactly in the order required during a manually operated
GENCCPM session. The last command can be followed by a carriage
return and the command:

        A>GENSYS

to end the command sequence and generate the system. If the GENSYS
command is not present, GENCCPM queries the console for changes.

The following example illustrates the use of the GENCCPM input file.
Assuming that the input file file specification is GENCCPM.IN, use
the following command to invoke GENCCPM:

    A>GENCCPM <GENCCPM.IN

Figure 2-10 shows a typical GENCCPM command file:

```
VERBOSE=N DESTDRIVE=D:
SYSPARAMS
OSSTART=4000 NPDESCS=20 QBUFSIZE=4FF TMPDRIVE=A: CMDLOGGING=Y
<cr>
MEMORY
DELETE=* ADD=2400,4000,400 ADD=4000,6000,800
<cr>
DISKBUFFERS
A:
8
4
4
2
hashing
*:        ; for all remaining drive questions
A:        ; share directory buffers with A:
A:        ; share data buffers with A:
hashing   ; hashing on all drives
<cr>
OSLABEL
Concurrent CP/M  Version 1.21  04/15/83
Hardware Configuration:
        A: 10 MB Hard Disk
        B: 5 MB Hard Disk
        C: Single-density Floppy
        D: Double-density Floppy
        M: Memory Disk
<cr>
GENSYS <cr> <------- Only if you do not want to be able
                      to specify additional changes
```

Figure 2-10. Typical GENCCPM Command File

After reading in the command file and optionally accepting any
additional changes you want to make, GENCCPM builds a system image
in the CCPM.SYS file in the manner described in Section 2.1.

<p align="center">End of Section 2</p>

Section 3
XIOS Overview

Concurrent CP/M Version 3.1, as implemented with one of the example XIOS's discussed in Section 3.1, is configured for operation with the Compu-Pro with at least two 8-inch floppy disk drives and at least 128K of RAM. All hardware dependencies are concentrated in subroutines collectively referred to as the Extended Input/Output System, or XIOS. You can modify these subroutines to tailor the system to almost any 8086 or 8088 disk-based operating environment. This section provides an overview of the XIOS, and variables and tables referenced within the XIOS.

The following material assumes that you are familiar with the CP/M-86 BIOS. To use this material fully, refer frequently to the example XIOS's found in source code form on the Concurrent CP/M distribution disk.

Note: Programs that depend upon the interface to the XIOS must check the version number of the operating system before trying direct access to the XIOS. Future versions of Concurrent CP/M can have different XIOS interfaces, including changes to XIOS function numbers and/or parameters passed to XIOS routines.

The XIOS must fit within the 64K System Data Segment along with the SYSDAT and Table Area. Concurrent CP/M accesses the XIOS through the two entry points INIT and ENTRY at offset 0C00H and 0C03H, respectively, in the System Data Segment. The INIT entry point is for system hardware initialization only. The ENTRY entry point is for all other XIOS functions. Because all operating system routines use a Call Far instruction to access the XIOS through these two entry points, the XIOS function routines must end with a Return Far instruction. Subsequent sections describe the XIOS entry points and other fixed data fields.

3.1 XIOS Header

The XIOS Header contains variables that GENCCPM uses when constructing the CCPM.SYS file and that the operating system uses when executing. Figure 3-1 illustrates the XIOS header.

COOH	JMP INIT				JMP ENTRY			SYSDAT	
C08H	SUPERVISOR				TICK	TICKS _SEC		DOOR	RESER- VED
C10H	NPCNS	NVCNS	NCCB	NLCB	CCB			LCB	
C18H	DPH(A)		DPH(B)		DPH(C)			DPH(D)	
C20H	DPH(E)		DPH(F)		DPH(G)			DPH(H)	
C28H	DPH(I)		DPH(J)		DPH(K)			DPH(L)	
C30H	DPH(M)		DPH(N)		DPH(O)			DPH(P)	
C38H	ALLOC								

Figure 3-1. XIOS Header

Table 3-1. XIOS Header Data Fields

Data Field	Explanation
JMP INIT	XIOS Initialization Point. At system boot, the Supervisor module executes a CALL FAR instruction to this location in the XIOS (XIOS Code Segment: 0C00H). This call transfers control to the XIOS INIT routine, which initializes the XIOS and hardware, then executes a RETURN FAR instruction. The JMP INIT instruction must be present in the XIOS.A86 file. For details of the INIT routine see Section 3.2, "INIT Entry Point."
JMP ENTRY	XIOS Entry Point. All access to the XIOS functions goes through the XIOS Entry Point. The operating system executes a far call (CALLF) to this location in the XIOS (XIOS Code Segment: 0C03H) whenever I/O is needed. This instruction transfers control to the XIOS ENTRY routine which calls the appropriate function within the XIOS. Once the function is complete, the ENTRY routine executes a return far (RETF) to the operating system. The RETF instruction must be present in the XIOS.A86 file. For details of the ENTRY routine, see Section 3.3, "XIOS ENTRY."

Table 3-1. (continued)

Data Field	Explanation
SYSDAT	The segment address of SYSDAT. It is in the Code Segment of the XIOS to allow access to data in SYSDAT while in interrupt routines and other areas of code where the Data Segment is unknown. For example, the following routine accesses the current process's Process Descriptor:

```
DSEG
            ORG 68H             ; point to RLR field
                                ; of SYSDAT
RLR         RW      1           ; does not generate
                                ; a hex value
            CSEG                ; of XIOS

            PUSH DS             ; Save XIOS Data
                                ; Segment
            MOV DS,CS:SYSDAT    ; Move the SYSDAT
                                ; segment address
                                ; into DS
            MOV BX,RLR          ; Move the current
                .               ; process's PD
                .               ; Address into BX
                .               ; and perform
                .               ; operation.  (See
                .               ; Fig 1-5 for expla-
                                ; nation of RLR)
            POP DS              ; Restore the XIOS
                                ; Data Segment
```

This variable is initialized by GENCCPM.

Data Field	Explanation
SUPERVISOR	FAR Address (double-word pointer) of the Supervisor Module entry point. Whenever the XIOS makes a system call, it must access the operating system through this entry point. GENCCPM initializes this field. Section 3.8, "XIOS System Calls", describes XIOS register usage and restrictions.

Table 3-1.   (continued)

Data Field	Explanation
TICK	Set Tick Flag Boolean. The Timer Interrupt routine uses this variable to determine whether the DEV_SETFLAG system call should be called to set the TICK_FLAG. Initialize this variable to zero (00H) in the XIOS.CON file. Concurrent CP/M sets this field to 0FFH whenever a process is delaying. The field is reset to zero (00H) when all processes finish delaying. See the Concurrent CP/M Operating System Programmer's Reference Guide for details on the DEV_SETFLAG and P_DELAY system calls. See Section 7 of this manual, "XIOS TICK Interrupt Routine," for more information on the XIOS usage of TICK.
TICKS_SEC	Number of Ticks per Second. This field must be initialized in the XIOS.CON file to be the number of ticks that make up one second as implemented by this XIOS. GENCCPM copies this field into the SYSDAT DATA. Application programmers can use TICKS_SEC to determine how many ticks to delay in order to delay one second. See Section 7, "XIOS TICK Interrupt Routine," for more information.
DOOR	Global Door Open Interrupt Flag. This field must be set to 0FFH by the drive door open interrupt handler routine if the XIOS detects that any drive door has been opened. The BDOS checks this field before every disk operation to verify that the media is unchanged. If a door has been opened, the XIOS must also set the Media Flag in the DPH associated with the drive.
NPCNS	Number of Physical Consoles. Initialize this field to the number of physical consoles, or user terminals connected to the system. This number does not include extra I/O devices. GENCCPM uses this value, and creates a PIN process for each physical console. It also copies NPCNS into the XPCNS field of the SYSDAT DATA.
NVCNS	Number of Virtual Consoles. Initialize this field to the number of virtual consoles supported by the XIOS in the XIOS.CON file. GENCCPM creates a TMP and a VOUT process for each virtual console. GENCCPM copies NVCNS into the NVCNS field of the SYSDAT DATA.

Table 3-1. (continued)

Data Field	Explanation
NCCB	Number of Logical Consoles. Initialize this field to the number of virtual consoles plus the number of character I/O devices supported by the XIOS. Character I/O devices are devices accessed through the console system calls of Concurrent CP/M (functions whose mnemonic begins with C_) but whose console numbers are beyond the range of the virtual consoles. Application programs access the character I/O devices by setting their default console number to the character I/O device's console number and using the regular console system calls of Concurrent CP/M. See the C_SET system call as described in the Concurrent CP/M Operating System Programmer's Reference Guide. GENCCPM copies this field into the NCCB field of the SYSDAT DATA.
NLCB	Number of List Control Blocks. Initialize this field in the XIOS.CON file to equal the number of list devices supported by the XIOS. A list device is an output-only device, typically a printer. GENCCPM copies this field into the NLCB field of the SYSDAT DATA.
CCB	Offset of the Console Control Block Table. Initialize this field in the XIOS.CON file to be the address of the CCB Table in the XIOS. A CCB Entry in the Table must exist for each of the consoles indicated in NCCB. Each entry in the CCB Table must be initialized as described in Section 4.1, "Console Control Block". GENCCPM copies this field into the CCB field of the SYSDAT DATA.
LCB	Offset of the List Control Block. This field is initialized in the XIOS.CON file to be the address of the LCB Table in the XIOS. There must be an LCB Entry for each of the list devices indicated in NLST. Each entry must be initialized as described in Section 4.3, "List Device Functions." GENCCPM copies this field into the LCB field of the SYSDAT DATA.

Table 3-1.   (continued)

Data Field	Explanation
DPH(A)-DPH(P)	Offset of initial Disk Parameter Header (DPH) for drives A through P, respectively. If the value of this field is 0000H, the drive is not supported by the XIOS. GENCCPM uses the DPH Table to initialize specific fields in the DPHs when it automatically creates BCBs and buffers. If the relevant DPH fields are not initialized to 0FFFFH, GENCCPM assumes the BCBs and buffers are defined by data already initialized in the XIOS.
ALLOC	This value is initialized in the XIOS to the size, in paragraphs, of an uninitialized RAM buffer area to be reserved for the XIOS by GENCCPM. When GENCCPM creates the CCPM.SYS image, it sets this field in the CCPM.SYS file to the starting paragraph (segment value) of the XIOS uninitialized buffer area. This value may then be used by the XIOS for based or indexed addressing into the buffer area. Typically, the XIOS uses this buffer area for the virtual console screen maps, programmable function key buffers, and nondisk-related I/O buffering. GENCCPM allocates this uninitialized RAM immediately following the system image and any system disk data or directory hashing buffers. Because the XIOS buffer area is not included in the CCPM.SYS file, it can be of any desired size without affecting system load time performance. If the ALLOC field is initialized to zero in the XIOS.CON file, GENCCPM allocates no buffer RAM and leaves ALLOC set to zero in the system image.

Listing 3-1 illustrates the XIOS Header definition:

```
;****************************************************
;*
;*      XIOS Header Definition
;*
;****************************************************

        CSEG
        org     0C00h

        jmp init        ;system initialization
        jmp entry       ;xios entry point

sysdat          dw      0       ;Sysdat Segment
supervisor      rw      2

        DSEG
        org     0C0Ch

tick            db      false           ;tick enable flag
ticks_sec       db      60              ;# of ticks per second
door            db      0               ;global drive door open
                                        ;       interrupt flag
rsvd            db      0               ;reserved for operating
                                        ;system use

npcns           db      4               ;number of physical consoles
nvcns           db      8               ;number of virtual consoles
nccb            db      8               ;total number of ccbs
nlst            db      1               ;number of list devices

ccb             dw      offset ccb0     ;offset of the first ccb
lcb             dw      offset lcb0     ;offset of first lcb

                                ;disk parameter header offset table

dph_tbl         dw      offset dph0     ;drive A:
                dw      offset dph1     ;B:
                dw      0,0,0           ;C:,D:,E:
                dw      0,0,0           ;F:,G:,H:
                dw      0,0,0           ;I:,J:,K:
                dw      0               ;L:
                dw      offset dph2     ;M:
                dw      0,0,0           ;N:,O:,P:
alloc           dw      0

;-----------------------------------------------------------------
```

Listing 3-1.   XIOS Header Definition

3.2 INIT Entry Point

The XIOS initialization routine entry point, INIT, is at offset 0C00H from the beginning of the XIOS code module. The INIT process calls the XIOS Initialization routine during system initialization. The sequence of events from the time CCPM.SYS is loaded into memory until the RSPs are created is important for understanding and debugging the XIOS.

The loader loads CCPM.SYS into memory at the absolute Code Segment location contained in the CCPM.SYS file Header, and initializes the CS and DS registers to the Supervisor code segment and the SYSDAT, respectively. At this point, the loader executes a JMPF to offset 0 of the CCPM.SYS code and begins the initialization code of the Concurrent CP/M SUP module as described below.   When loading CCPM.SYS under DDT-86 or SID-86, use the R command and set the code and data segments manually before beginning execution.   You cannot use the E command because it initializes the data segment base page to incorrect values. See Section 8, "Debugging the XIOS."

1. The first step of initialization in the SUP is to set up the INIT process. The INIT process performs the rest of system initialization at a priority equal to 1.

2. The INIT process calls the initialization routines of each of the other modules with a Far Call instruction.   The first instruction of each code module is assumed to be a JMP instruction to its initialization routine.   The XIOS initialization routine is the last of these modules called. Once this call is made, the XIOS initialization code is never used again. Thus, it can be located in a directory buffer or other uninitialized data area.

3. As shown in the example XIOS listing, the initialization routine must initialize all hardware and interrupt vectors. Interrupt 224 is saved by the SUP module and restored upon return from the XIOS. Because DDT-86 uses interrupts 1, 3, and 225, do not initialize them when debugging the XIOS with DDT-86 running under CP/M-86.   On each context switch, interrupt vectors 0, 1, 3, 4, 224, and 225 are saved and restored as part of a process's environment.

4. The XIOS initialization routine can optionally print a message to the console before it executes a Far Return (RETF) instruction upon completion. Note that each TMP prints out the string addressed by the VERSION variable in the SYSDAT DATA. This string can be changed using the OSLABEL Menu in GENCCPM.

5. Upon return from the XIOS, the SUP Initialization routine, running under the INIT process, creates some queues and starts up the RSPs. Once this is done, the INIT process terminates.

The XIOS INIT routine should initialize all unused interrupts to
vector to an interrupt trap routine that prevents spurious
interrupts from vectoring to an unknown location. The example XIOS
handles uninitialized interrupts by printing the name of the process
that caused the interrupt followed by an uninitialized interrupt
error message. Then the interrupting process is unconditionally
terminated.

Concurrent CP/M saves Interrupt Vector 224 prior to system
initialization and restores it following execution of the XIOS INIT
routine. However, it does not store or alter the Non-Maskable
Interrupt (NMI) vector, INT 2. Setting NMI is also the
responsibility of the XIOS. The example XIOS first initializes all
the Interrupt Vectors to the uninitialized interrupt trap, then
initializes specifically used interrupts.

Note: When debugging the XIOS with DDT-86 running under CP/M-86,
do not initialize Interrupt Vectors 1, 3, and 225. The example
XIOS's have a debug flag that is tested by the INIT routine for this
purpose.

3.3 XIOS ENTRY

All accesses to the XIOS after initialization go through the ENTRY
routine. The entry point for this routine is at offset 0C03H from
the beginning of the XIOS code module. The operating system
accesses the ENTRY routine with a Far Call to the location offset
0C03H bytes from the beginning of the SYSDAT Segment. When the XIOS
function is complete, the ENTRY routine returns by executing a Far
Return instruction, as in the example XIOS's. On entry, the AL
register contains the function number of the routine being accessed,
and registers CX and DX contain arguments passed to that routine.
The XIOS must maintain all segment registers through the call. This
means that the CS, DS, ES, SS, and SP registers are maintained by
the functions being called.

Table 3-2. XIOS Register Usage

Registers on Entry
AL = function number BX = PC-MODE parameter CX = first parameter DX = second parameter DS = SYSDAT segment ES = User Data Area AH, SI, DI, BP, DX, CX are undefined
Registers on Return
AX = return or XIOS error code BX = AX DS = SYSDAT segment ES = User Data Area SI, DI, BP, DX, CX are undefined

All XIOS functions, with the exception of disk functions, use the register conventions shown above.

The segment registers (DS and ES) must be preserved through the ENTRY routine. However, when calling the SUP from within the XIOS, the ES Register must equal the UDA of the running process and DS must equal the System Data Segment. Thus, if the XIOS is going to perform a string move or other code using the ES Register, it must preserve ES using the stack as in the following example:

```
    push es
    mov es,segment_address
     ...
    rep movsw
     ...
    pop es
```

In the example XIOS's, the XIOS function routines are accessed through a function table with the function number being the actual table entry. Table 3-3 lists the XIOS function numbers and the corresponding XIOS routines; detailed explanations of the functions appear in the referenced sections of this document. Listing 3-2 is an example XIOS ENTRY Jump Table.

Table 3-3. XIOS Functions

Function Number	XIOS Routine
Console Functions -- Section 4.2	
Function 0 IO_CONST CONSOLE STATUS Function 1 IO_CONIN CONSOLE INPUT Function 2 IO_CONOUT CONSOLE OUTPUT Function 7 IO_SWITCH SWITCH SCREEN Function 8 IO_STATLINE DISPLAY STATUS LINE	
List Device Functions -- Section 4.3	
Function 3 IO_LSTST LIST STATUS Function 4 IO_LSTOUT LIST OUTPUT	
Other Character Devices -- Section 4.4	
Function 5 IO_AUXIN AUXILIARY INPUT Function 6 IO_AUXOUT AUXILIARY OUTPUT	
Poll Device Function -- Section 4.5	
Function 13 IO_POLL POLL DEVICE	
Disk Functions -- Section 5.1	
Function 9 IO_SELDSK SELECT DISK Function 10 IO_READ READ DISK Function 11 IO_WRITE WRITE DISK Function 12 IO_FLUSH FLUSH BUFFERS Function 35 IO_INT13_READ READ DOS DISK Function 36 IO_INT13_WRITE WRITE DOS DISK	
PC Mode Character Functions -- Section 6	
Function 30 IO_SCREEN GET/SET SCREEN Function 31 IO_VIDEO VIDEO IO Function 32 IO_KEYBD KEYBOARD MODE Function 33 IO_SHFT SHIFT STATUS Function 34 IO_EQCK EQUIPMENT CHECK	

```
;--------------------------------------------------------------
;                      XIOS FUNCTION TABLE
;--------------------------------------------------------------

functab dw      io_const        ; 0 - console status
        dw      io_conin        ; 1 - console input
        dw      io_conout       ; 2 - console output
        dw      io_listst       ; 3 - list status
        dw      io_list         ; 4 - list output
        dw      io_auxin        ; 5 - aux in
        dw      io_auxout       ; 6 - aux out
        dw      io_switch       ; 7 - switch screen
        dw      io_statline     ; 8 - display status line
        dw      io_seldsk       ; 9 - select disk
        dw      io_read         ;10 - read sector
        dw      io_write        ;11 - write sector
        dw      io_flushbuf     ;12 - flush buffer
        dw      io_poll         ;13 - poll device
        dw      io_ret          ;14 - dummy return
        dw      io_ret          ;15 - dummy return
        dw      io_ret          ;16 - dummy return
        dw      io_ret          ;17 - dummy return
        dw      io_ret          ;18 - dummy return
        dw      io_ret          ;19 - dummy return
        dw      io_ret          ;20 - dummy return
        dw      io_ret          ;21 - dummy return
        dw      io_ret          ;22 - dummy return
        dw      io_ret          ;23 - dummy return
        dw      io_ret          ;24 - dummy return
        dw      io_ret          ;25 - dummy return
        dw      io_ret          ;26 - dummy return
        dw      io_ret          ;27 - dummy return
        dw      io_ret          ;28 - dummy return
        dw      io_ret          ;29 - dummy return
        dw      io_screen       ;30 - get/set screen mode
        dw      io_video        ;31 - video i/o
        dw      io_keybd        ;32 - keyboard info
        dw      io_shft         ;33 - shift status
        dw      io_eqck         ;34 - equipment check
        dw      io_intl3_read   ;35 - read DOS disk
        dw      io_intl3_write  ;36 - write DOS disk

;--------------------------------------------------------------
```

Listing 3-2. XIOS Function Table

3.4 Converting the CP/M-86 BIOS

The implementation of Concurrent CP/M described below assumes that you have written and fully debugged a CP/M-86 BIOS on the target Concurrent CP/M machine. This is desirable for the following reasons:

- The implementation of CP/M-86 on the target Concurrent CP/M machine greatly simplifies debugging the XIOS using DDT-86 or SID-86.

- A CP/M-86 or a running Concurrent CP/M system is required for the initial generation of the Concurrent CP/M system when using GENCCPM.

- You can use the CP/M-86 BIOS as a basis for construction of the target Concurrent CP/M XIOS.

To transform the CP/M-86 BIOS to the Concurrent CP/M XIOS, you must make the following principal changes. Details of the changes given in the following list can be found in the referenced sections of this manual, and in the example XIOS's found on the Concurrent CP/M distribution disk. Often it is easier to start with the example Concurrent CP/M XIOS and replace the hardware-dependent code with the corresponding drivers from the existing CP/M-86 BIOS. However, there are several important changes, also outlined below, that you must make to the CP/M-86 drivers before they work in the Concurrent CP/M XIOS.

1. Change the BIOS Jump Table to use only the two XIOS entry points, INIT and ENTRY. Concurrent CP/M assumes these entry points to be unconditional jump instructions to the corresponding routines. The INIT routine takes the place of the CP/M-86 cold start entry point and is only invoked once, at system initialization time. The ENTRY routine is the single entry point indexing into all XIOS functions and replaces the BIOS Jump Table. Concurrent CP/M accesses the ENTRY routine with the XIOS function number in the AL register. The example XIOS then uses the value in the AL register as an index into a function table to obtain the address of the corresponding function routine.

2. Add a SUP module interface routine to enable the XIOS to execute Concurrent CP/M system calls. The XIOS is within the operating system area and already uses the User Data Area stack; therefore, the XIOS cannot make system calls in the conventional manner. See Section 3.8, "XIOS System Calls."

3. Modify the console routines to reflect the IO_CONST, IO_CONIN, IO_CONOUT, IO_LSTST, and IO_LISTOUT specifications. Note that the register conventions for Concurrent CP/M are different from CP/M-86 and MP/M-86.

4. Rewrite the CP/M-86 disk routines to conform to the IO_SELDSK, IO_READ, IO_WRITE, and IO_FLUSH specifications.

5. Change all polled devices to use the Concurrent CP/M DEV_POLL system call. See Sections 4.5, "IO_POLL Function"; 3.5, "Polled Devices"; and Section 6 of the _Concurrent CP/M Operating System Programmer's Reference Guide_.

6. Change all interrupt-driven device drivers to use the Concurrent CP/M DEV_WAITFLAG and DEV_SETFLAG system calls. See Sections 3.6, "Interrupt Devices"; 7, "XIOS Tick Interrupt Routine"; and Section 6 of the _Concurrent CP/M Operating System Programmer's Reference Guide_.

7. Change the structure of the Disk Parameter Header (DPH) and Disk Parameter Block (DPB) data structures referenced by the XIOS disk driver routines. See Sections 5.4, "Disk Parameter Header" and 5.5, "Disk Parameter Block."

8. Remove the Blocking/Deblocking algorithms from the XIOS disk drivers. The Concurrent CP/M BDOS now handles the blocking/deblocking function. The XIOS still handles sector translation.

9. Change the disk routines to reference the Input/Output Parameter Block (IOPB) on the stack. See Section 5.2, "IOPB Data Structure." Modify the disk driver routine to handle multisector reads and writes.

10. Rewrite the console and list driver code to handle virtual consoles and, possibly, multiple physical consoles. Details of the virtual console system are given in Section 4, "Character Devices."

11. Implement the TICK interrupt routine (see I_TICK in the example XIOS's). This routine is used for process dispatching, maintaining the P_DELAY system call, and waking up the CLOCK process RSP. See Section 7, "XIOS Tick Interrupt Routine."

3.5 Polled Devices

Polled I/O device drivers in the CP/M-86 BIOS typically execute a
small compute-bound instruction loop waiting for a ready status from
the I/O device.   This causes the driver routine to spend a
significant portion of CPU execution time looping.   To allow other
processes use of the CPU resource during hardware wait periods, the
Concurrent CP/M XIOS must use a system call, DEV_POLL, to place the
polling process on the Poll List.   After the DEV_POLL call, the
dispatcher stops the process and calls the XIOS IO_POLL function
every dispatch until IO_POLL indicates the hardware is ready.   The
dispatcher then restores the polling process to execution and the
process returns from the DEV_POLL call.   Since the process calling
the DEV_POLL function does not remain in ready state, the CPU
resource becomes available to other processes until the I/O hardware
is ready.

To do polling, a process executing an XIOS function calls the
Concurrent CP/M DEV_POLL system call with a poll device number.   The
dispatcher then calls the XIOS IO_POLL function with the same poll
device number.   The example XIOS uses the poll device number to
index into a table of poll routine entry points, calls the
appropriate poll function and returns the I/O device status to the
dispatcher.

3.6 Interrupt Devices

As in the case of polled I/O devices, an XIOS driver handling an
interrupt-driven I/O device should not execute a wait loop or halt
instruction while waiting for an interrupt to occur.

The Concurrent CP/M XIOS handles interrupt-driven devices by using
DEV_WAITFLAG and DEV_SETFLAG system calls.   A process that needs to
wait for an interrupt to occur makes a DEV_WAITFLAG system call with
a flag number.   The system stops this process until the desired XIOS
interrupt handler routine makes a DEV_SETFLAG system call with the
same flag number.   The waiting process then continues execution.
The interrupt handler follows the steps outlined below, executing a
far jump (JMPF) to the Dispatcher entry point.   The interrupt
handler can also perform an IRET instruction when it is done.
However, jumping directly to the Dispatcher gives a little faster
response to the process waiting on the flag, and is logically
equivalent to the IRET instruction.

If interrupts are enabled within an interrupt routine, a TICK
interrupt can cause the interrupt handler to be dispatched.   This
dispatch could make interrupt response time unacceptable.   To avoid
this situation, do not re-enable interrupts within the interrupt
handlers or only jump to the dispatcher when not in another
interrupt handler routine.

Interrupt handlers under Concurrent CP/M differ from those in an
8080 environment due to machine architecture differences. Study the
TICK interrupt handler in the example XIOS's carefully.   During
initial debugging, it is not recommended that interrupts be
implemented until after the system works in a polled environment.
An XIOS interrupt handler routine must perform the following basic
steps:

1. Do a stack switch to a local stack. The interrupted process
   might not have enough stack space for a context save.

2. Save the register environment of the interrupted process, or at
   least the registers that will be used by the interrupt routine.
   Usually the registers are saved on the local stack established
   in step (1) above.

3. Satisfy the interrupting condition. This can include resetting
   the hardware and performing a DEV_SETFLAG system call to notify
   a process that the interrupt for which it was waiting has
   occurred.

4. Restore the register environment of the interrupted process.

5. Switch back to the original stack.

6. Either a Jump Far (JMPF) to the dispatcher or an Interrupt
   Return (IRET) instruction must be executed to return from the
   interrupt routine. Note the above discussion on which return
   method to use for different situations. Usually, when
   interrupts are not re-enabled within the interrupt handler, a
   Jump Far (JMPF) to the dispatcher is executed on each system
   tick and after a DEV_SETFLAG call is made. Otherwise, if
   interrupts are re-enabled an IRET instruction is executed.

Note: DEV_SETFLAG is the only Concurrent CP/M system call an
interrupt routine may call. This is because the DEV_SETFLAG call is
the only system call the operating system assumes has no process
context associated with it.   DEV_SETFLAG must enter the operating
system through the SUP entry point at SYSDAT:0000H and cannot use
INT 224.

3.7   8087 Exception Handler

The default for the Concurrent CP/M system is to provide no support
for the 8087 co processor. This section explains what must be done
to provide support for the 8087 chip. To support the 8087 the XIOS
initialization code must initialize some fields in the SYSDAT area.
The XIOS must also contain a default exception handler to handle any
interrupts from the 8087.   The system is structured so that a
programmer can write an individual exception handler for the 8087.

The XIOS initialization code must first check for the presence of
the 8087 chip by using the FNINIT instruction. If it is present,
the following fields in SYSDAT must be set up:

    SEG_8087,OFF_8087   Must be set to the segment and offset of
                                    the 8087 interrupt vector.

    SYS_87_S6,
    SYS_87_OF   Must be set to the segment and offset of
                                    the XIOS default exception handler.

    OWNER_8087   Must be set to 0 to indicate that there
                                  is an 8087 present in the system.   The
                                  Default value is FFFFH which indicates
                                  no 8087. FFFFH is put in this field by
                                  the SUP initialization code.

The 8087 interrupt vector must also be set to the segment and offset
of the XIOS default exception handler.

Any exception handler for the 8087 must perform its functions in a
certain order to guarantee program integrity in a multitasking
environment. The following is an outline of the example default
8087 exception handler. See Listing 3-3 for the code of the
example.

1. Save the 8086 environment.

2. Save the 8087 environment.

3. Clear the 8087 IR (status word).

4. Disable 8087 interrupts.

5. Acknowledge the interrupt (hardware dependent).

6 Look at the owner_8087 field, and perform the desired action.
   Note that 8086 interrupts are currently off. Do not perform
   any action that would turn them back on yet. The default
   exception handler uses the OWNER_8087 field to terminate the
   process on a severe error.

7. Restore the 8086 environment.

8. Restore the 8087 environment with clear status.   This re-
   enables the 8087 interrupts.

9. Execute an IRET instruction to return and re-enable the 8086
   interrupts.

If the 8087 environment is not restored before 8086 interrupts are
enabled and an interrupt occurs (for example, 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 will be brought back into 8086 context and will write
over the new 8087 context.

All 8087 processes are initialized by the system with the address of
the default exception handler. If a process wants to use its own
exception handler, it must initially overwrite the 8087 interrupt
vector with the address of its own exception handler. On each
context switch, the 8087 interrupt vector is saved and restored as
part of the 8087 process's environment.

The hardware-dependent address of the 8087 interrupt vector is
provided in the SEG_8087 and OFF_8087 fields of the system data
area.

An individual exception handler must follow the same sequence of
events described for the default handler. Failure to do so will
have unpredictable results on the system. If possible, make this
default interrupt handler re-entrant.

ndpint:

```
;=====================================
;  8087 Default Exception Handler
;=====================================
;
;   This is the example default exception handler.
;   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 in
;   response to these conditions.
;   This handler ignores non-severe errors (overflow, etc.)
;   and terminates 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            ; Stack switch for 8086 env
        mov     ndp_spreg,sp
        mov     ss,sysdat
        mov     sp,offset ndp_tos       ; Save 8086 registers
        push    ax
        push    bx
        push    cx
        push    dx
        push    di
        push    si
        push    bp
        push    es
        mov     es,sysdat               ; Now save 8087 env
        FNSTENV env_8087                ; Save 8087 Process Info
        FWAIT
        FNCLEX                          ; Clear 8087 interrupt reque:
        xor     ax,ax
        FNDISI                          ; Disable 8087 interrupts

        mov     al,020h                 ; Send int ack's - 1 for sla'
        out     060h,al
        mov     al,020h                 ; - 1 for master PIC
        out     058h,al

        call    in_8087                 ; Check 8087 error condition
                                        ; if error is severe,
                                        ; process will abort

        mov     bx,offset env_8087      ; clear 8087 status word
        mov     byte ptr 2[bx],0        ; for env restore
```

Listing 3-3. 8087 Exception Handler

```
        pop     es                   ; Restore 8086 env.
        pop     bp
        pop     si
        pop     di
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        mov     ss,ndp_ssreg         ; Switch to previous stack
        mov     sp,ndp_spreg
        FLDENV  env_8087             ; Restore 8087 environment
        FWAIT                        ; with good status
        pop     ds                   ; Restore previous data segment
        iret

in_8087:

        mov     bx,owner_8087        ; Get the Process Descriptor
        test    bx,bx                ; Check if owner has
        jz      end_87               ; already terminated
        mov     si,offset env_8087   ; If severe error, terminate
        mov     ax,statusw[si]       ; If not, return and continue
        test    ax,03ah              ; 3A = under/overflow, precision,
        jnz     end_87               ;       and denormalized operand
        or      p_flag[bx],080h      ; Must be zero divide or invalid
                                     ; operation (stack error)
                                     ; Turn on terminate flag

end_87:
        ret

;==================================
```

Listing 3-3.   (continued)

3.8   XIOS System Calls

Routines in the XIOS cannot make system calls in the conventional
manner of executing an INT 224 instruction. The conventional entry
point to the SUP does a stack switch to the User Data Area (UDA) of
the current process.   The XIOS is considered within the operating
system, and a process entering the XIOS is already using the UDA
stack.   Therefore, a separate entry point is used for internal
system calls.

Location 0003H of the SUP code segment is the entry point for
internal system calls. Register usage for system calls through this
entry point is similar to the conventional entry point. They are as
follows:

```
        Entry:    CX = System call number
                  DX = Parameter
                  DS = Segment address if DX is an offset to a
                        structure
                  ES = User Data Area
        Return:   AX = BX = Return
                  CX = Error Code
                  ES = Segment value if system call returns
                        an offset and segment.  Otherwise
                        ES is unaltered and equals the UDA
                        upon return.
                  DX, SI, DI, BP are not preserved.
```

The only differences between the internal and user entry points are
the CX and ES registers on entry. For the internal call, CH must
always be 0. ES must always point to the User Data Area of the
current process. The UDA segment address can be obtained through
the following code:

```
        org 68H

        rlr     rw     1       ; ready list root
                                ; in SYSDAT

        org (XIOS code segment)

        mov si,rlr
        mov es,10h[si]
```

Note: On entry to the XIOS, ES is equal to the UDA segment
address. The ES Register must equal the UDA on return from any XIOS
function called by the XIOS ENTRY routine. Interrupt routines must
restore ES and any other altered registers to their value upon entry
to the routine, before performing an IRET instruction or a JMPF to
the dispatcher.

<p align="center">End of Section 3</p>

Section 4
Character Devices

This section describes the XIOS functions necessary for Character I/O. Some additional functions, described in Section 6, are needed to run DOS programs.

Concurrent CP/M treats all serial I/O devices as consoles. Serial I/O devices are divided into two categories: virtual consoles and extra I/O devices. Each virtual console is assigned to a specific physical console or user terminal. Associated with each serial I/O device (virtual console or extra I/O device) is a Console Control Block (CCB). The serial I/O devices and CCBs are numbered relative to zero. Each process contains, in its Process Descriptor, the number of its default console. The default console can be either a virtual console or an extra serial I/O device.

Concurrent CP/M can be configured in a number of different ways by changing the CCB table in the XIOS. It can be configured for one or more user terminals (physical consoles), and extra I/O devices. The number of virtual consoles assigned to each user terminal is set in the CCB table. Up to 256 serial I/O devices can be implemented, depending on the specific application.

The XIOS header defines the size and location of the CCB table. In the header, the CCB field points to the beginning of the CCB table. The NCCB field contains the number of entries in the CCB table. The NVCNS field tells how many of the CCBs are virtual consoles. See "XIOS Header" in Section 3 for more information.

The XIOS might or might not maintain a buffer containing the screen contents and cursor position for each virtual console, depending on how the system is to appear to the user. Keep in mind that this buffer can be over 4K bytes per virtual console. Practical considerations of memory space might require keeping the number of virtual consoles reasonably small if buffers are maintained. Also note that if the user terminals are connected to serial ports, the time to update the screen for a screen switch can be up to 2 seconds. One example XIOS has eight virtual consoles, divided among multiple serial terminals.

By convention, the first NVCNS serial I/O devices are the virtual
consoles. The NVCNS parameter is located in the XIOS Header. The
XPCNS field tells how many user terminals there are. XPCNS must be
less than or equal to NVCNS. XPCNS does not include extra I/O
Devices. Consoles beyond the last virtual console represent other
serial I/O devices. When a process makes a console I/O call with a
console number higher than the last virtual console, it references
the Console Control Block for the called device number. Therefore a
CCB for each serial I/O device is absolutely necessary.

List Devices under Concurrent CP/M are output-only. The XIOS must
reserve and initialize a List Control Block for each list output
device. When a process makes a list device XIOS call, it references
the appropriate LCB.

4.1 Console Control Block

A Console Control Block Table must be defined in the XIOS. There
must be one CCB for each virtual console and Character I/O device
supported by the XIOS, as indicated by the NCCB variable in the XIOS
Header. The table must begin at the address indicated by the CCB
variable in the XIOS Header.

```
┌──────────┐        ┌────────────────┐
│   CCB    │        │     CCB 0      │     (virtual console 0)
│  (XIOS   │        └────────────────┘
│  Header) │               .
└──────────┘               .
                           .
                    ┌────────────────┐
                    │  CCB NVCNS-1   │     (last virtual console)
                    ├────────────────┤
                    │   CCB NVCNS    │     (first extra char-
                    └────────────────┘        acter I/O device)
                           .
                           .
                           .
                    ┌────────────────┐
                    │  CCB NCCB-1    │     (last extra char-
                    └────────────────┘        acter I/O device)
```
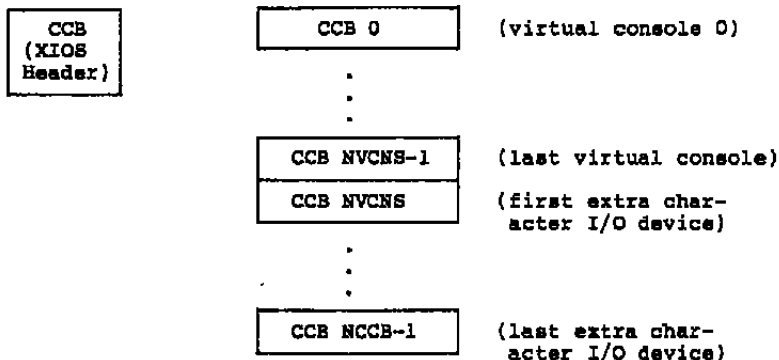
Figure 4-1. The CCB Table

The number of CCBs used for virtual consoles equals the NVCNS field
in the XIOS Header. Any additional CCB entries are used for other
character devices to be supported by the XIOS. The CCB entries are
numbered starting with zero to match their logical console device
numbers. Therefore, the last CCB in the CCB Table is the (NCCB-1)th
CCB.

Each CCB corresponding to a virtual console has several fields which
must be initialized, either when the XIOS is assembled or by the
XIOS INIT routine.   These fields allow you to choose the
configuration of the virtual consoles. The PC field indicates the
physical console this virtual console is assigned to. The VC field
is the virtual console number. This number must be unique within
the system.   The LINK field points to the CCB of the next virtual
console assigned to this physical console. The last virtual console
assigned to each physical console should have the LINK field set to
zero (0000H). Figure 4-2 shows a diagram of the CCBs for a system
with two physical consoles, with three and two virtual consoles
assigned respectivly. For CCBs outside the virtual console range
corresponding to extra I/O devices, these fields must all be
initialized   to   zero   (00H), except for the PC field.   Also,
initialize to zero (00H) all fields marked RESERVED in Figure 4-3.

```
      ┌────────┬──────┬──────┐
      │ CCB 0  │ PC 0 │ VC 0 │
      ├────────┴──────┴──────┤
   ┌──│ LINK                 │
   │  └──────────────────────┘
   │
   └─►┌────────┬──────┬──────┐
      │ CCB 1  │ PC 0 │ VC 1 │
      ├────────┴──────┴──────┤
   ┌──│ LINK                 │
   │  └──────────────────────┘
   │
   └─►┌────────┬──────┬──────┐
      │ CCB 2  │ PC 0 │ VC 2 │
      ├────────┴──────┴──────┤
 0◄───│ LINK                 │
      └──────────────────────┘

      ┌────────┬──────┬──────┐
      │ CCB 3  │ PC 1 │ VC 3 │
      ├────────┴──────┴──────┤
   ┌──│ LINK                 │
   │  └──────────────────────┘
   │
   └─►┌────────┬──────┬──────┐
      │ CCB 4  │ PC 1 │ VC 4 │
      ├────────┴──────┴──────┤
 0◄───│ LINK                 │
      └──────────────────────┘
```
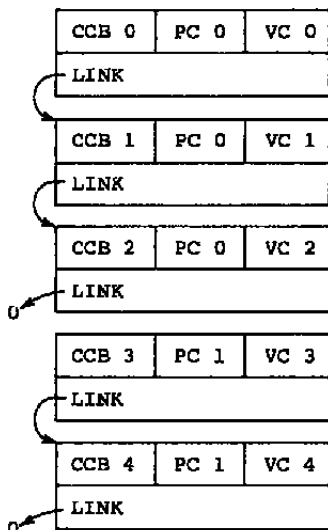
Figure 4-2.   CCBs for Two Physical Consoles

00	OWNER				RESERVED		
08h	MIMIC		PC	VC	RESERVED		STATE
10h	MAXBUFSIZE			RESERVED			
18h	RESERVED						
20h	RESERVED						
28h	LINK		RESERVED				

Figure 4-3. Console Control Block Format

Table 4-1. Console Control Block Data Fields

Data Field	Explanation
OWNER	Address of the Process Descriptor of the process that currently owns the virtual console or character I/O device. This field is used by the XIOS Status Line Function (IO_STATLINE) to find the name of the current owner. Initialize this field display to zero (0000H). If the value in this field is zero when Concurrent CP/M is running, no process owns the device.
MIMIC	This field indicates which list device receives the characters typed on the virtual console when the CTRL-P command is in effect. MIMIC must be initialized to OFFH. Note that this list device is not necessarily the same as the default list device indicated in the Process Descriptor whose address is in the OWNER field of the CCB. Consider the following interaction at the console:

Table 4-1. (continued)

Data Field	Explanation

	A>printer The TMP's PD has a 0 in
	     its LIST field.
	Printer Number = 0
	A>^P Printer echo to list
	     device 0.
	A>printer 2 The TMP's PD has a 2 in
	     its LIST field.
	Printer Number = 2
	A>pip lst:=letter.prn LETTER.PRN is sent to
	     list device 2 Printer
	     echo is still going to
	     list device 0, echoing
	     the last two commands.
	
	     The example status line
	     routine distinguishes
	     between the default
	     list device and the
	     CTRL-P list device by
	     displaying
	
	     Printer=2
	
	     for the default list
	     device, and
	
	     ^P=0
	
	     after the last command
	     in the illustration
	     above.

PC	Physical console number.
VC	Virtual console number. Virtual console numbers must be unique within the system.

Table 4-1.   (continued)

Data Field	Explanation
STATE	The least significant bit of this field indicates the background mode of the virtual console. The XIOS Status Line Function routine uses this information to display the background mode for the current foreground console. This bit has the following values:
	0 background is dynamic 1 background is buffered
	The STATE field can be initialized to 0 or 1 on each virtual console to specify the background mode at system startup. The Concurrent CP/M VCMODE utility allows the user to change the background mode.
MAXBUFSIZE	The MAXBUFSIZE field indicates the maximum size of the buffer file used to store characters when a background virtual console is in buffered mode. When a virtual console is placed in background mode by the user, a temporary file is created on the temporary drive, containing console output sent to the virtual console. These files are named VOUTx.$$$, where x equals the number of the associated virtual console. The MAXBUFSIZE field is the maximum size to which this file can grow. If this maximum is reached, the drive is Read-Only, or there is no more free space on the drive, subsequent console output causes the background process attached to the virtual console to be stopped. The MAXBUFSIZE parameter is in Kilobytes and must be initialized in the XIOS CCB entries. The Concurrent CP/M VCMODE utility allows the user to change this value. The legal range for MAXBUFSIZE is 1 to 8191 decimal (1FFFH).
LINK	Address of the next CCB assigned to the same physical console. Zero (0000H) if this is the last or only virtual console for this physical console.

4.2 Console I/O Functions

A major difference between the Concurrent CP/M XIOS and the CP/M-86
BIOS drivers is how they wait for an event to occur. In CP/M-86, a
routine typically goes into a hard loop to wait for a change in
status of a device, or executes a Halt (HLT) instruction to wait for
an interrupt. In Concurrent CP/M, this does not work. It can be of
some use, however, during the very early stages of debugging the
XIOS.

Basically, two ways to wait for a hardware event are used in the
XIOS. For noninterrupt-driven devices, use the DEV_POLL method.
For interrupt-driven devices, use the DEV_SETFLAG/DEV_FLAGWAIT
method. These are both ways in which a process waiting for an
external event can give up the CPU resource, allowing other
processes to run concurrently. For detailed explanations of the
DEV_POLL, DEV_FLAGWAIT and DEV_SETFLAG system calls, see Section 6
of the Concurrent CP/M Operating System Programmer's Reference
Guide.

```
                IO_CONST    CONSOLE INPUT STATUS



            Return the Input Status of the specified
                    Serial I/O Device.


        Entry Parameters:
            Register   AL:   00H (0)
                       DL:   Serial I/O Device Number

        Returned   Value:
            Register   AL:   0FFH if character ready
                       0     if no character ready
                       BL:   Same as AL
                       ES, DS, SS, SP  preserved
```

The IO_CONST routine returns the input status of the specified
character I/O device. This function is only called by the operating
system for console numbers greater than NVCNS-1, in other words,
only for devices which are not virtual consoles. If the status
returned is 0FFH, then one or more characters are available for
input from the specified device.

```
┌─────────────────────────────────────────────────────────┐
│                                                         │
│          IO_CONIN    CONSOLE INPUT                      │
│                                                         │
├─────────────────────────────────────────────────────────┤
│                                                         │
│        Return a character from the console              │
│          keyboard or a serial I/O device.               │
│                                                         │
├─────────────────────────────────────────────────────────┤
│   Entry Parameters:                                     │
│        Register  AL:  01H (1)                           │
│                  DL:  Serial I/O Device Number          │
│                                                         │
│   Returned   Value:                                     │
│        Register  AH:  00H if returning                  │
│                           character data                │
│                  AL:  character                         │
│                                                         │
│                  AH:  0FFH if returning a               │
│                           switch screen request         │
│                  AL:  virtual console requested         │
│                                                         │
│                  BX:  same as AX in all cases           │
│                  ES, DS, SS, SP  preserved              │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

Because Concurrent CP/M supports the full 8-bit ASCII character set, the parity bit must be masked off from input devices which use it. However, it should not be masked off if valid 8-bit characters are being input.

You choose the key or combination of keys that represent the virtual consoles by the implementation of IO_CONIN. One of the example XIOS's uses the function keys f1 through f3 to represent the virtual consoles assigned to each user terminal.

IO_CONIN must check for PC-MODE. PC-MODE is enabled whenever DOS programs are running. It is enabled or disabled by the IO_KEYBD (Function 32) call. If PC-MODE is enabled, all function keys are passed through to the calling process. If it is disabled, function keys that do not have an associated XIOS function are usually ignored on input. See Section 6.2 "Keyboard Functions" for information on the IO_KEYBD call.

IO_CONOUT   CONSOLE OUTPUT

Display and/or output a character to the
         specified device.

Entry Parameters:
    Register AL: 02H (2)
              CL: Character to send
              DL: Virtual console to send to

Returned   Value: NONE

                ES, DS, SS, SP preserved

The XIOS might or might not buffer background virtual consoles,
depending on the user interface desired, memory constraints, and
methods of updating the terminals. This section describes how the
example XIOS's handle virtual consoles.

The example XIOS's buffer all virtual consoles.   All virtual
consoles have a screen image area in RAM. This image reflects the
current contents of the screen, both characters and attributes.
Each screen image is contained in a separate segment.

Each virtual console also has a Screen Structure associated with it.
This structure contains the segment address of the screen image, the
cursor location (offset in the segment), and any other information
needed for the screen. This structure can be expanded to support
additional hardware requirements, such as color CRTs.

For a screen-buffered implementation, when a character is given to
IO_CONOUT, it performs the following operations:

1. Look up the screen structure for this virtual console and get
   the segment address of the screen image.

2. Update the image, including all changes caused by escape
   sequences. This could involve changes to the characters on the
   screen (clear screen), the cursor location (home), or the
   attributes of the individual characters (inverse video).

3. If this console is in the foreground and on a serial terminal,
   put the character out to the physical terminal. This requires
   looking up the true physical console number.

When a process calls this function with a device number higher than
the last virtual console number, the character should be sent
directly to the serial device that the CCB represents.

Note that for screen buffering it is necessary to buffer 25 lines
when in PC-MODE, but only 24 lines otherwise. The PC-MODE flag is
set by Function 32, which is described in Section 6.2.

```
+-------------------------------------------------+
|                                                 |
|         IO_SWITCH    SWITCH SCREEN              |
|                                                 |
+-------------------------------------------------+
|                                                 |
|     Place the current virtual console into the  |
|        background and the specified virtual     |
|          console into the foreground.           |
|                                                 |
+-------------------------------------------------+
|  Entry Parameters:                              |
|     Register  AL:   07H (7)                      |
|               DL:   Virtual Console # to         |
|                     switch to                    |
|                                                 |
|     Return  Values:   NONE                       |
|                                                 |
|                  ES, DS, SS, SP  preserved       |
|                                                 |
+-------------------------------------------------+
```

When IO_SWITCH is called, the XIOS copies the screen image in memory
to the physical screen. It must move the cursor on the physical
screen to the proper position for the new foreground console.

IO_SWITCH is responsible for doing a flagset to restart a background
process that is waiting to go into graphics mode. If the process's
screen is to be switched into the foreground, do a flagset on the
flag that was used by IO_SCREEN to flagwait the process. See
Section 6.1 for more information on IO_SCREEN.

IO_SWITCH will be implemented differently for machines with video
RAM (such as the IBM Personal Computer) and serial terminals. For
IBM Personal Computers, the screen switch can be done by doing a
block move from the screen image to the video RAM, and a physical
cursor positioning. A serial terminal must be updated by sending a
character at a time, with insertion of escape sequences for the
attribute changes.

Concurrent CP/M calls IO_SWITCH only when there is no process currently in the XIOS performing console output to either the foreground virtual console being switched out, or the background virtual console being switched into the foreground. Therefore, the XIOS never has to update a screen while simultaneously switching it from foreground to background, or vice versa.

One of the example IO_SWITCH routines performs the following operations:

1. Get the screen structure and image segment for the new virtual console.

2. Find the physical console number for this virtual console.

3. If this is a video-mapped console, save the current display by doing a block move. If it is a serial terminal, clear the physical screen and home the cursor.

4. If this is a video-mapped display, do a block move of the new screen image to the video RAM, and re-position the cursor. If it is a serial terminal, send each character to the physical screen. Check each character's attribute byte, and send any escape sequences necessary to display the characters with the correct attributes.

```
┌─────────────────────────────────────────────────────┐
│                                                       │
│        IO_STATLINE    DISPLAY STATUS LINE             │
│                                                       │
├─────────────────────────────────────────────────────┤
│                                                       │
│     Display specified text on the status line         │
│                                                       │
├─────────────────────────────────────────────────────┤
│                                                       │
│    Entry Parameters:                                  │
│       Register AL:   08H (8)                           │
│                CX:   if 0000H, continue to             │
│                      update the normal                │
│                      status line                      │
│                      if CX = offset, print            │
│                      string at DX:CX                  │
│                      if 0FFFFH, resume normal         │
│                      status line display              │
│       Register DL:   physical console to display      │
│                      status line on (if CX = 0)       │
│       Register DX:   segment address of               │
│                      optional string (if CX <> 0)     │
│                                                       │
│    Return Values:    NONE                             │
│                      ES, DS, SS, SP  preserved        │
│                                                       │
└─────────────────────────────────────────────────────┘
```

When IO_STATLINE is called with CX = 0, the normal status
information is displayed by IO_STATLINE on the physical console
specified in DL. The normal status line typically consists of the
foreground virtual console number, the state of the foreground
virtual console, the process that owns the foreground virtual
console, the removable-media drives with open files, whether control
P, S, or O are active, and the default printer number.   The
IO_STATLINE function in the example XIOS's display some of the above
information. Usually when IO_STATLINE is called, DL is set to the
physical console to display the status line on. You must translate
this to the current (foreground) virtual console before getting the
information for the status line (such as the process owning the
console). The status line can be modified, expanded to any size, or
displayed in a different area than the status line implemented in
the example XIOS's. A common addition to the status line is a time-
of-day clock.

A status line is strongly recommended. However, if there are only
24 lines on the display device, you might choose not to implement a
status line. In this case IO_STATLINE can just return when called.

The normal status line is updated once per second by the CLOCK RSP.
If there is more than one user terminal connected to the system,
this update occurs once per second on a round-robin basis among the
physical terminals. Thus, if four terminals are connected each one
is updated every four seconds by the CLOCK.

The operating system also requests normal status line updates when
screen switches are made and when control P, S or O change state.
The XIOS might call IO_STATLINE from other routines when some value
displayed by the status line changes.

Note: IO_STATLINE's re-entrancy depends in part on having separate
buffers for each physical console.

The IO_STATLINE routine should not display the status line on a user
terminal that is in graphics mode.   It should check the same
variable as IO_SCREEN (Function 30).   IO_SCREEN is described in
Section 6.1 "Screen I/O Functions".

IO_STATLINE also should not display on a console that is in PC-MODE.
Check the variable set by Function 32 to see if a console is in PC-
MODE. See Section 6.2 for information on Function 32.

Most calls to IO_STATLINE to update the status line have DL set to
the physical terminal that is to be updated. When IO_STATLINE is
called with CX not equal to 0000H or 0FFFFH, then CX is assumed to
be the byte offset and DX the paragraph address of an ASCII string
to print on the status line. This special status line remains on
the screen until another special status line is requested, or
IO_STATLINE is called with CX=0FFFFH. While a special status line
is being displayed, calls to IO_STATLINE with CX=0000H are ignored.
When IO_STATLINE function is called with CX=0FFFFH, the normal
status line is displayed and subsequent calls with CX=0000H cause
the status line to be updated with current information.

When IO_STATLINE is called to display a special status line, DL does
not contain the physical console number. The physical console
number can be obtained by the following method:

1. Get the address of SYSDAT

2. Look at the RLR (Ready List Root). The first process on the
   list is the current process.

3. Look at the Process Descriptor (pointed to by RLR). The p_cns
   field contains the virtual console number of the current
   process. See the Concurrent CP/M Operating System Programmer's
   Reference Guide for a description of the Process Descriptor.

4. Look up the CCB for this virtual console and find the physical
   console number in it.

A process calling IO_STATLINE with a special status line (DX:CX =
address of the string) must call IO_STATLINE before termination with
CX=0FFFFH. Otherwise the normal status line is never shown again.
There is no provision for a process to find out which status line is
being displayed.

4.3 List Device Functions

A List Control Block (LCB), similar to the CCB, must be defined in
the XIOS for each list output device supported. The number of LCBs
must equal the NLCB variable in the XIOS Header. The LCB Table
begins with LCB zero, and ends with LCB NLCB-1, according to their
logical list device names.

```
┌──────────┐        ┌─────────────┐
│   LCB    │───────▶│   LCB  0    │        (LIST DEVICE 0)
│  (XIOS   │        └─────────────┘
│  HEADER) │               •
└──────────┘               •
                           •
                   ┌─────────────┐
                   │ LCB NLCB-1  │        (LAST LIST DEVICE)
                   └─────────────┘
```
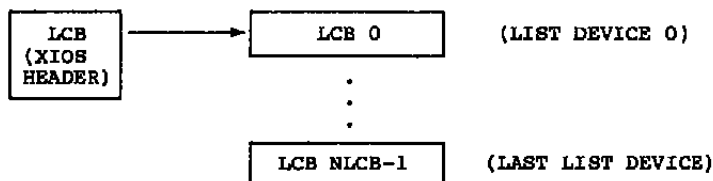
Figure 4-4.   The LCB Table

4-13

00H	OWNER		R E S E R V E D
08H	RESER- VED	M- SOURCE	

Figure 4-5. List Control Block (LCB)

Table 4-2. List Control Block Data Fields

Field	Explanation
OWNER	Address of the PD of the process that currently owns the List Device. If no process currently owns the list device, then OWNER=0. If OWNER=0FFFFH, this list device is mimicking a console device that is in CTRL-P mode.
MSOURCE	If OWNER=0FFFFH, MSOURCE contains the number of the console device this list device is mimicking; otherwise MSOURCE = 0FFH. **Note:** MSOURCE must be initialized to 0FFH. All other LCB fields must be initialized to 0.

```
              IO_LSTST    LIST STATUS


            Return List Output Status


  Entry Parameters:
      Register  AL:  03H (3)
                DL:  List Device number

  Returned   Value:
      Register  AL:  0FFH if Device Ready
                0    if Device Not Ready
                BL:  Same as AL

                ES, DS, SS, SP  preserved
```

The IO_LSTST function returns the output status of the specified
list device.

```
┌─────────────────────────────────────────────────────────────┐
│                                                               │
│              IO_LSTOUT     LIST OUTPUT                         │
│                                                               │
├─────────────────────────────────────────────────────────────┤
│                                                               │
│     Output Character to Specified List Device                 │
├─────────────────────────────────────────────────────────────┤
│    Entry Parameters:                                          │
│         Register  AL:   04H (4)                               │
│                   CL:   Character                             │
│                   DL:   List Device number                    │
│                                                               │
│    Returned   Value:   None                                   │
│                                                               │
│                        ES, DS, SS, SP   preserved             │
│                                                               │
└─────────────────────────────────────────────────────────────┘
```

The IO_LSTOUT function sends a character to the specified List
Device. List device numbers start at 0. It is the responsibility
of the XIOS device driver to zero the parity bit for list devices
that require it.

4.4 Auxiliary Device Functions

These XIOS functions are accessible only through the Concurrent CP/M
S_BIOS system call. Software that uses this call can access the AUX:
device by placing the appropriate parameters in the Bios Descriptor.
For further information, see the Concurrent CP/M Operating System
Programmer's Reference Guide under the S_BIOS system call.

If you choose not to implement the AUX: device then the IO_AUXOUT
function can simply return, while IO_AUXIN should return a character
26 (1AH), CTRL-Z, indicating end of file.

```
+-------------------------------------------------------+
|                                                       |
|          IO_AUXIN    AUXILIARY INPUT                  |
|                                                       |
+-------------------------------------------------------+
|                                                       |
|    Input a character from the Auxiliary Device        |
|                                                       |
+-------------------------------------------------------+
|  Entry Parameters:                                    |
|      Register  AL:  05H (5)                            |
|                                                       |
|  Returned   Value:                                    |
|      Register  AL:  Character                          |
|                                                       |
|               ES, DS, SS, SP  preserved               |
|                                                       |
+-------------------------------------------------------+
```

```
+-------------------------------------------------------+
|                                                       |
|                                                       |
|          IO_AUXOUT   AUXILIARY OUTPUT                 |
|                                                       |
+-------------------------------------------------------+
|                                                       |
|   Output a character to the Auxiliary Device          |
|                                                       |
+-------------------------------------------------------+
|  Entry Parameters:                                    |
|      Register  AL:  06H (6)                            |
|                CL:  Character                          |
|                                                       |
|  Returned   Value:  None                              |
|                                                       |
|               ES, DS, SS, SP  preserved               |
|                                                       |
+-------------------------------------------------------+
```

4.5 IO_POLL Function

```
┌─────────────────────────────────────────────────────┐
│                                                       │
│              IO_POLL      POLL DEVICE                  │
│                                                       │
├─────────────────────────────────────────────────────┤
│                                                       │
│     Poll Specified Device and Return Status           │
├─────────────────────────────────────────────────────┤
│                                                       │
│     Entry Parameters:                                 │
│           Register AL:  0DH (13)                      │
│                   DL:   Poll Device Number            │
│                                                       │
│     Returned   Value:                                 │
│           Register AL:  0FFH if ready                 │
│                         0    if not ready             │
│                   BL:   Same as AL                    │
│                   ES, DS, SS, SP  preserved           │
│                                                       │
└─────────────────────────────────────────────────────┘
```

The IO_POLL function interrogates the status of the device indicated
by the poll device number and returns its current status. It is
called by the dispatcher.

A process polls a device only if the Concurrent CP/M DEV_POLL system
call has been made. The poll device number used as an argument for
the DEV_POLL system call is the same number that the IO_POLL
function receives as a parameter. Typically only the XIOS uses
DEV_POLL. The mapping of poll device numbers to actual physical
devices is maintained by the XIOS. Each polling routine must have a
unique poll device number. For instance, if the console is polled,
it must have different poll device numbers for console input and
console output.

The sample XIOS's show the IO_POLL function taking the poll device
number as an index to a table of poll functions. Once the address
of the poll routine is determined, it is called and the return
values are used directly for the return of the IO_POLL function.

<div align="center">End of Section 4</div>

Section 5
Disk Devices

In Concurrent CP/M, a disk drive is any I/O device that has a directory and is capable of reading and writing data in 128-byte logical sectors. The XIOS can therefore treat a wide variety of peripherals as disk drives if desired. The logical structure of a Concurrent CP/M disk drive is presented in detail in Section 10, "OEM Utilities." CP/M can also support PC-DOS and MS-DOS disks. The term DOS refers to both PC-DOS and MS-DOS.

This section discusses the Concurrent CP/M XIOS disk functions, their input and output parameters, associated data structures, and calculation of values for the XIOS disk tables.

5.1 Disk I/O Functions

Concurrent CP/M performs Disk I/O with a single XIOS call to the IO_READ or IO_WRITE functions. These functions reference disk parameters contained in an Input/Output Parameter Block (IOPB), which is located on the stack, to determine which disk drive to access, the number of physical sectors to transfer, the track and sector to read or write, and the DMA offset and segment address involved in the I/O operation. See Section 5.2, "IOPB Data Structure." Prior to each IO_READ or IO_WRITE call, the BDOS initializes the IOPB.

If a physical error occurs during an IO_READ or IO_WRITE operation, the function routine should perform several retries (10 is recommended) to attempt to recover from the error before returning an error condition to the BDOS.

The Disk I/O routine interfaces in the Concurrent CP/M XIOS are quite different from those in the CP/M-86 BIOS. The SETTRK, SETSEC, SETDMA, and SETDMAB XIOS functions no longer exist because IO_READ or IO_WRITE have absorbed their functions. WBOOT, HOME, SECTRAN, GETSEGB, GETIOB, and SETIOB are not used by any routines outside the I/O system, and so have been dropped. Also, hard loops within the disk routines must be changed to make either DEV_POLL or DEV_WAITFLAG system calls. See Sections 3.5, "Polled Devices"; 4.5, "IO_POLL Function"; and 3.6, "Interrupt Devices." For initial debugging, Concurrent CP/M runs with the CP/M-86 BIOS physical sector read and write routines, with the addition of an IOPB-referencing routine, multisector read/write capability, and modification to handle the new DPH and DPB structures. Once the system runs well, all hard loops should be changed to either DEV_POLL or DEV_WAITFLAG system calls. See also the discussion in Sections 3.5 and 3.6 of this manual.

IO_SELDSK   SELECT DISK

Select the specified Disk Drive

Entry Parameters: AL: 09H (9)
                     CL: Disk Drive Number
                     DL: (bit 0): 0 if first select

Return Values: AX: offset of DPH if no error
                     AX: 00H if invalid drive
                     BX: Same as AX
                     ES, DS, SS, SP preserved

The IO_SELDSK function checks if the specified disk drive is valid
and returns the address of the corresponding Disk Parameter Header
if the drive is valid. The specified disk drive number is 0 for
drive A, 1 for drive B, up to 15 for drive P. On each disk select,
IO_SELDSK must return the offset of the selected drive's Disk
Parameter Header relative to the SYSDAT segment address.

If there is an attempt to select a nonexistent drive, IO_SELDSK
returns 00H in AL as an error indicator. Although IO_SELDSK must
return the Disk Parameter Header (DPH) address for the specified
drive on each call, postpone the actual physical disk select
operation until an I/O function, IO_READ or IO_WRITE, is performed.
This is due to the fact that disk select operations can take place
without a subsequent disk operation and thus disk access might be
substantially slower using some disk controllers.

IO_SELDISK must return a DPH containing the address of the Disk
Parameter Block (DPB). The DPB must be properly formatted to
reflect the type of media supported by the selected drive. On a
first time select, this function must determine if this disk is a
CP/M disk, or a DOS disk. For CP/M media, return a regular DPB.
For a DOS disk return an extended DPB. See Section 5.5 "Disk
Parameter Block" for more information on the two DPB formats. See
Section 5.8 "Multiple Media Support" for more information on
generating a system that supports both types of disks.

On entry to IO_SELDSK, you can determine whether it is the first
time the specified disk has been selected. Register DL, bit 0
(least significant bit), is a zero if the drive has not been
previously selected. This information is of interest in systems
that read configuration information from the disk to dynamically set
up the associated DPH and DPB. See Section 5.8 "Multiple Media
Support". If Register DL, bit 0, is a one, IO_SELDSK must return a
pointer to the same DPH as it returned on the initial select.

```
+------------------------------------------------------------+
|                                                            |
|               IO_READ    READ  SECTOR                      |
|                                                            |
+------------------------------------------------------------+
|                                                            |
|          Read sector(s)  defined by the IOPB               |
|                                                            |
+------------------------------------------------------------+
|                                                            |
|   Entry Parameters:  IOPB filled in (on stack)             |
|            Register AL:  0AH (10)                           |
|                                                            |
|      Return Values:  AL:   0 if no error                   |
|                            1 if physical error             |
|                         OFFH if media density              |
|                                has changed                 |
|                      AH:   Extended error  code            |
|                            (Table 5-1)                     |
|                      BL:  Same as AL                       |
|                      BH:  Same as AH                       |
|                      ES, DS, SS, SP  preserved             |
|                                                            |
+------------------------------------------------------------+
```

The IO_READ Function transfers data from disk to memory according to
the parameters specified in the IOPB. The disk Input/Output
Parameter Block (IOPB), located on the stack, contains all required
parameters, including drive, multisector count, track, sector, DMA
offset, and DMA segment, for disk I/O operations. See Section 5.2,
"IOPB Data Structure." If the multisector count is equal to 1, the
XIOS should attempt a single physical sector read based upon the
parameters in the IOPB. If a physical error occurs, the read
function should return a 1 in AL and BL, and the appropriate
extended error code in AH and BH. The XIOS should attempt several
retries (10 recommended) before giving up and returning an error
condition.

For disk drivers with auto density select, IO_READ should
immediately return OFFH if the hardware detects a change in media
density. The BDOS then performs an IO_SELDSK system call for that
drive, reinitializing the drive's parameter tables in order to avoid
writing erroneous data to disk.

If the multisector count is greater than 1, the IO_READ routine is
required to read the specified number of physical sectors before
returning to the BDOS. The IO_READ routine should attempt to read
as many physical sectors as the specified drive's disk controller
can handle in one operation. Additional calls to the disk
controller are required when the disk controller cannot transfer the
requested number of sectors in a single operation. If a physical
error occurs during a multisector read, the read function should
return a 1 in AL and BL and the appropriate extended error code in
AH and BH.

If the disk controller hardware can only read one physical sector at
a time, the XIOS disk driver must make the number of single
physical-sector reads defined by the multisector count.   In any
case, when more than one call to the controller is made, the XIOS
must increment the sector number and add the number of bytes in each
physical sector to the DMA address for each successive read. If,
during a multisector read, the sector number exceeds the number of
the last physical sector of the current track, the XIOS has to
increment the track number and reset the sector number to 0. This
concept is illustrated in Listing 5-1, part of a hard disk driver
routine.

In this example, if the multisector count is zero, the routine
returns with an error. Otherwise, it immediately calls the
read/write routine for the present sector and puts the return code
passed from it in AL. If there is no error, the multisector count
is decremented. If the multisector count now equals zero, the read
or write is finished and the routine returns. If not, the sector to
read or write is incremented. If, however, the sector number now
exceeds the number of sectors on a track (MAXSEC), the track number
is incremented and the sector number set to zero. The routine then
performs the number of reads or writes remaining to equal the
multisector count, each time adding the size of a physical sector to
the DMA offset passed to the disk controller hardware.

Table 5-1. Extended Error Codes

Code	Meaning
80H	Attachment failed to respond
40H	Seek operation failed
20H	Controller has failed
10H	Bad CRC
8H	DMA overrun
4H	Sector not found
3H	Write protect disk error
2H	Address mark not found
1H	Bad command

Listing 5-1 illustrates multisector operations:

```
;***************************************************
;*
;*      common code for hard disk read and write
;*
;***************************************************

hd_io:
        push es                 ;save UDA
        cmp mcnt,0              ;if multisector count = 0
        je hd_err              ;return error
hdiol:
            call iohost            ;read/write physical sector
            mov al,retcode         ;get return code
            or al,al               ;if not 0
            jnz hd_err             ;return error
            dec mcnt               ;decrement multisector count
            jz return_rw           ;if mcnt = 0 return
              mov ax,sector
              inc ax               ;next sector
              cmp ax,maxsecl jb same_trak  ;is sector < max sector
                inc track          ; no - next track
                xor ax,ax          ; initialize sector to 0
same_trak:
            mov sector,ax          ;save sector #
            add dmaoff,secsiz      ;increment dma offset by sector size
            jmps hdiol             ;read/write next sector
hd_err:
            mov al,1               ;return with error indicator
return_rw:
          pop es                  ;restore UDA
          ret                     ;return with error code in AL

;***************************************************
;* IOHOST performs the physical reads and writes to  *
;* the physical disk.                                 *
;***************************************************

iohost:
  ...
  ...
  ...

        ret

;-------------------------------------------------------------------
```

Listing 5-1. Multisector Operations

IO_INT13_READ   READ DOS SECTOR

Read DOS sector(s) defined by the IOPB

Entry Parameters: DOS IOPB filled in (on stack)
         Register AL: 23H (35)

Return Values: AL: 0 if no error
                      1 if physical error
                   OFFH if media density
                        has changed
                AH: Extended error code
                      (Table 5-1)
                BL: Same as AL
                BH: Same as AH
                ES, DS, SS, SP preserved

IO_INT13_READ emulates DOS's interrupt 13 read disk operation. It
reads a DOS disk as specified by the DOS format IOPB. It is used on
DOS media only. It operates like IO_READ except for the different
IOPB. The DOS IOPB is defined in Section 5.2

```
+-----------------------------------------------------------+
|                                                           |
|              IO_WRITE    WRITE SECTOR                      |
|                                                           |
+-----------------------------------------------------------+
|                                                           |
|          Write sector(s) defined by the IOPB              |
|                                                           |
+-----------------------------------------------------------+
|                                                           |
|   Entry Parameters:  IOPB filled in (on stack)            |
|            Register AL:  0BH (11)                          |
|                                                           |
|   Return Values:  AL:    0 if no error                    |
|                          1 if physical error              |
|                          2 if Read/Only Disk              |
|                       0FFH if media density               |
|                            has changed                    |
|                   AH:   Extended error code               |
|                          (Table 5-1)                      |
|                   BL:   Same as AL                        |
|                   BH:   Same as AH                        |
|                   ES, DS, SS, SP  preserved               |
|                                                           |
+-----------------------------------------------------------+
```

The IO_WRITE function transfers data from memory to disk according
to the parameters specified in the IOPB. This function works in
much the same way as the read function, with the addition of a
Read/Only Disk return code. IO_WRITE should return this code
when the specified disk controller detects a write-protected
disk.

```
┌─────────────────────────────────────────────────────────────┐
│                                                               │
│        IO_INT13_WRITE    WRITE DOS SECTOR                      │
│                                                               │
├─────────────────────────────────────────────────────────────┤
│                                                               │
│        Write DOS sector(s) defined by the IOPB                │
│                                                               │
├─────────────────────────────────────────────────────────────┤
│                                                               │
│   Entry Parameters:  DOS IOPB filled in (on stack)            │
│            Register AL:  24H (36)                              │
│                                                               │
│   Return Values:  AL:    0 if no error                        │
│                          1 if physical error                  │
│                          2 if Read/Only Disk                  │
│                        OFFH if media density                  │
│                             has changed                       │
│                   AH:   Extended error code                   │
│                         (Table 5-1)                           │
│                   BL:   Same as AL                            │
│                   BH:   Same as AH                            │
│                   ES, DS, SS, SP  preserved                   │
│                                                               │
└─────────────────────────────────────────────────────────────┘
```

IO_INT13_WRITE is similar to IO_WRITE. It uses a DOS IOPB, and
writes to a DOS disk. It emulates DOS's interrupt 13 write
function. The DOS IOPB is defined in Section 5.2.

```
┌─────────────────────────────────────────────────┐
│                                                 │
│         IO_FLUSH    FLUSH BUFFERS               │
│                                                 │
├─────────────────────────────────────────────────┤
│                                                 │
│    Write pending I/O system buffers to disk     │
│                                                 │
├─────────────────────────────────────────────────┤
│                                                 │
│   Entry Parameters:   Register AL: 0CH (12)     │
│                                                 │
│      Returned Value:                            │
│         Register AL:  0 if No Error             │
│                       1 if Physical Error       │
│                       2 if Read—Only Disk       │
│                  AH:  Extended error code       │
│                       (Table 5-1)               │
│                  BL:  Same as AL                │
│                  BH:  Same as AH                │
│                  ES, DS, SS, SP  preserved      │
│                                                 │
└─────────────────────────────────────────────────┘
```

The IO_FLUSH function indicates that all blocking/deblocking buffers
or disk—caching buffers used by the I/O system should be flushed,
written to the disk. This does not include the LRU buffers that are
managed by the BDOS. This function is called whenever a process
terminates, a file is closed or a disk drive is reset. The XIOS
must return the error codes for the IO_FLUSH function in register
AX, after 10 recovery attempts as described in the IO_READ function.

5.2 IOPB Data Structure

The purpose of this and the following sections is to present the
organization and construction of tables and data structures within
the XIOS that define the characteristics of the Concurrent CP/M disk
system. Since there is no Concurrent CP/M GENDEF utility, you must
code the XIOS DPHs and DPBs by hand, using values calculated from
the information presented below.

The disk Input/Output Parameter Block (IOPB) contains the necessary
data required for the IO_READ and IO_WRITE functions. IO_INT13_READ
and IO_INT13_WRITE use a variation of the IOPB called the DOS IOPB.
It is described at the end of this section. These parameters are
located on the stack, and appear at the example XIOS IO_READ and
IO_WRITE function entry points as described below. The IOPB example
in this section assumes that the ENTRY routine calls the read or
write routines through only one level of indirection; therefore, the
XIOS has placed only only one word on the stack. RETADR is reserved
for this local return address to the ENTRY routine. The XIOS disk
drivers may index or modify IOPB parameters directly on the stack,
since they are removed by the BDOS when the function call returns.
Typically, the IOPB fields are defined relative to the BP and SS
registers.   The first instruction of the IO_READ and IO_WRITE
routines sets the BP register equal to the SP register for indexing
into the IOPB. Listing 5-2 illustrates this.

+14	DRV	MCNT
+12	TRACK	
+10	SECTOR	
+8	DMASEG	
+6	DMAOFF	
+4	RETSEG	
+2	RETOFF	<== SP value at XIOS ENTRY
SP+0	RETADR	<== SP value at disk routines
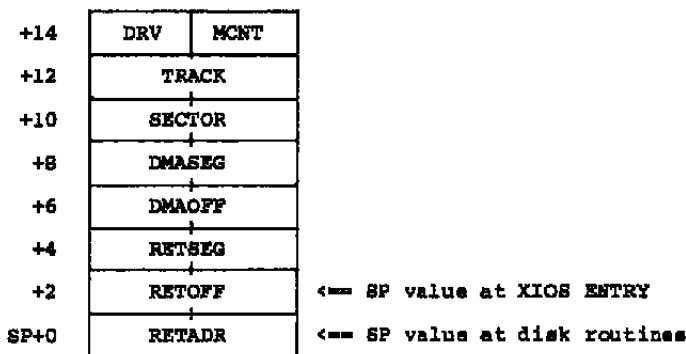
Figure 5-1.   Input/Output Parameter Block (IOPB)

Table 5-2.   IOPB Data Fields

Data Field	Explanation
DRV	Logical Drive Number. The Logical Drive Number specifies the logical disk drive on which to perform the IO_READ or IO_WRITE function. The drive number may range from 0 to 15, corresponding to drives A through P respectively.
MCNT	Multisector Count. To transfer logically consecutive disk sectors to or from contiguous memory locations, the BDOS issues an IO_READ or IO_WRITE function call with the multisector count greater than 1. This allows the XIOS to transfer multiple sectors in a single disk operation. The maximum value of the multisector count depends on the physical sector size, ranging from 128 with 128-byte sectors to 4 with 4096-byte sectors. Thus, the XIOS can transfer up to 16K directly to or from the DMA address in a single operation. For a more complete explanation of multisector operations, along with example code and suggestions for implementation within the XIOS, see Section 5.3, "Multisector Operations on Skewed Disks."
TRACK	Logical Track Number. The Track Number defines the logical track for the specified drive to seek. The BDOS defines the Track Number relative to 0, so for disk hardware which defines track numbers beginning with a physical track of 1, the XIOS needs to increment the track number before passing it to the disk controller.

Table 5-2.   (continued)

Data Field	Explanation
SECTOR	Sector Number. The Sector Number defines the logical sector for a read or write operation on the specified drive. The sector size is determined by the parameters PSH and PHM defined in the Disk Parameter Block. See Section 5.5. The BDOS defines the Sector Number relative to 0. For disk hardware that defines sector numbers beginning with a physical sector of 1, the XIOS will need to increment the sector number before passing it to the disk controller. If the specified drive uses a skewed-sector format, the XIOS must translate the sector number according to the translation table specified in the Disk Parameter Header.
DMASEG, DMAOFF	DMA Segment and Offset. The DMA offset and segment define the address of the data to transfer for the read or write operation. This DMA address may reside anywhere in the 1-megabyte address space of the 8086-8088 microprocessor. If the disk controller for the specified drive can only transfer data to and from a restricted address area, the IO_READ and IO_WRITE functions must block move the data between the DMA address and this restricted area before a write or following a read operation.
RETSEG,RETOFF	BDOS Return Segment and Offset. The BDOS return segment and offset are the Far Return address from the XIOS to the BDOS.
RETADR	Local Return Address. The local return address returns to the ENTRY routine in the example XIOS.

Listing 5-2 illustrates the IOPB definition, and how the IOPB is used in the IO_READ and IO_WRITE routines:

```
;*****************************
;*
;*     IOPB Definition
;*
;*****************************
;
;   Read and Write disk parameter equates
;
;   At the disk read and write function entries,
;    all disk I/O parameters are on the stack
;    and the stack at these entries appears as
;    follows:
;
;
;        +14    DRV    MCNT       Drive and Multisector count
;
;        +12         TRACK       Track number
;
;        +10        SECTOR       Physical sector number
;
;        +8       DMA_SEG        DMA segment
;
;        +6       DMA_OFF        DMA offset
;
;        +4       RET_SEG        BDOS return segment
;
;        +2       RET_OFF        BDOS return offset
;
;       SP+0      RET_ADR        Local ENTRY return address
;                                 (assumes one level of call
;                                  from ENTRY routine)
;
;   These parameters can be indexed and modified
;    directly on the stack and will be removed
;    by the BDOS after the function is complete

drive    equ      byte ptr 14[bp]
mcnt     equ      byte ptr 15[bp]
track    equ      word ptr 12[bp]
sector   equ      word ptr 10[bp]
dmaseg   equ      word ptr 8[bp]
dmaoff   equ      word ptr 6[bp]

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

Listing 5-2. IOPB Definition

```
;=========
IO_READ:          ; Function 11: Read sector
;=========
; Reads the sector on the current disk, track and
; sector into the current DMA buffer.
;        entry:  parameters on stack
;        exit:   AL = 00 if no error occurred
;                AL = 01 if an error occurred

        mov bp,sp                 ;set BP for indexing into IOPB
          .
          .
          .
        ret

;===========
IO_WRITE:         ; Function 12: Write disk
;===========
; Write the sector in the current DMA buffer
; to the current disk on the current
; track in the current sector.
;        entry:  CL = 0 - Deferred Writes
;                     1 - non-deferred writes
;                     2 - def-wrt 1st sect unalloc blk
;        exit:   AL = 00H if no error occurred
;                   = 01H if error occurred
;                   = 02H if read only disk

        mov bp,sp                 ;set BP for indexing into IOPB
          .
          .
          .
        ret

;----------------------------------------------------------------
```

Listing 5-2. (continued)

Figure 5-2 shows the DOS IOPB used by IO_INT13_READ and
IO_INT13_WRITE. It is similar to the regular IOPB. The DOS IOPB
fields are defined in Table 5-3.

+14	DRV	MCNT
+12	TRACK	HEAD
+10	SECTOR	00
+8	DMASEG	
+6	DMAOFF	
+4	RETSEG	
+2	RETOFF	<== SP value at XIOS ENTRY
SP+0	RETADR	<== SP value at disk routines

Figure 5-2. DOS Input/Output Parameter Block (IOPB)

Table 5-3. DOS IOPB Data Fields

Data Field	Explanation
TRACK	Track or cylinder number. This number must be in the range 0 - 39.
HEAD	Head number. This number must be 0 or 1.
SECTOR	Sector number. This number must be in the range 1 - 8.
	All other DOS IOPB data fields are the same as the regular IOPB defined in Table 5-2.

5.3 Multisector Operations on Skewed Disks

On many implementations of older Digital Research operating systems, disk performance is improved through sector skewing. This technique logically numbers the sectors on a track such that they are not sequential. An example of this is the standard Digital Research 8-inch disk format, where the sectors are skewed by a factor of 6. The following discussion illustrates how to optimize disk performance on skewed disks with multisector I/O requests.

Concurrent CP/M-86 supports multiple-sector read and write operations at the XIOS level to minimize rotational latency on block disk transfers. You must implement the multiple-sector I/O facility in the XIOS by using the multisector count passed in the IOPB.

When the disk format uses a skew table to minimize rotational latency for single-record transfers, it is more difficult to optimize transfer time for multisector operations. One method of doing this is to have the XIOS read/write function routine translate each logical sector number into a physical sector number. Then it creates a table of DMA addresses with each sector's DMA address indexed into the table by the physical sector number.

As a result, the requested sectors are sorted into the order in which they physically appear on the track. This allows all of the required sectors on the track to be transferred in as few disk rotations as possible. The data from each sector must be separately transferred to or from its proper DMA address. If during a multisector data transfer the sector number exceeds the number of the last physical sector of the current track, the XIOS will have to increment the track number and reset the sector number to 0. It can then complete the operation for the balance of sectors specified in the IO_READ or IO_WRITE function call. See the example accompanying the IO_READ function.

SECTOR INDEXES	PHYSICAL ASSOCIATED DMA ADDRESS
00	DMA_ADDR_0
01	DMA_ADDR_1
.	.
.	.
.	.
N	DMA_ADDR_N

Figure 5-3. DMA Address Table for Multisector Operations

If an error occurs during a multisector transfer, the XIOS should
return the error immediately to terminate the read or write BDOS
function call.

In Listing 5-3, common read/write code for an XIOS disk driver, the
routine gets the DPH address by calling the IO_SELDSK function. It
checks to verify a nonzero DPH address, and returns if the address
is invalid (zero). Then the disk parameters are taken from the DPH
and DPB and stored in local variables. Once the physical record
size is computed from DPB values, the DMA address table can be
initialized. The INITDMATBL routine fills the DMA address table
with 0FFFFH word values. The size of the DMA table equals one word
greater than the number of sectors per track, in case the sectors
index relative to 1 for that particular drive. If the multisector
count is zero, the routine returns an error. Otherwise, the sector
number is compared to the number of sectors per track to determine
if the track number should be incremented and the sector number set
to zero. If this is the case, the sectors for the current track are
transferred, and the DMA address table is reinitialized before the
next tracks are read or written.

The current sector number is moved into AX and a check is made on
the translation table offset address. If this value is zero, no
translation table exists and translation is not performed; The
sector number is translated and used to index into the DMA address
table. The current DMA address, incremented by the physical sector
size if a multisector operation, is stored in the table for use by
the RW_SECTS routine. Local values, beginning with i, are
initialized for the various parameters needed by the disk hardware,
and the disk driver routine is called.

Listing 5-3 illustrates multisector unskewing:

```
;*****************************************************
;*
;*      DISK I/O EQUATES
;*
;*****************************************************

xlt     equ     0       ;translation table offset in DPH
dpb     equ     8       ;disk parameter block offset in DPH
spt     equ     0       ;sectors per track offset in DPB
psh     equ     15      ;physical shift factor offset in DPB

;*****************************************************
;*
;*      DISK I/O CODE AREA
;*
;*****************************************************

;
read_write:     ;unskews and reads or writes multisectors
;-----------
;       input:  SI = read or write routine address
;       output: AX = return code

        mov cl,drive
        mov dl,1
        call seldsk                     ;get DPH address
        or bx,bx! jnz dsk_ok            ;check if valid
ret_error:
            mov al,1                    ; return error if not
            ret
dsk_ok:
        mov ax,xlt[bx]
        mov xltbl,ax                    ;save translation table address
        mov bx,dpb[bx]
        mov ax,spt[bx]
        mov maxsec,ax                   ;save maximum sector per track
        mov cl,psh[bx]
        mov ax,128
        shl ax,cl                       ;compute physical record size
        mov secsiz,ax                   ; and save it
        call initdmatbl                 ;initialize dma offset table
        cmp mcnt,0
        je ret_error
```

Listing 5-3. Multisector Unskewing

```
rw_l:
        mov ax,sector            ;is sector < max sector/track
        cmp ax,maxsec! jb same_trk
          call rw_sects          ; no - read/write sectors on track
          call initdmatbl        ; reinitialize dma offset table
          inc track              ; next track
          xor ax,ax
          mov sector,ax          ; initialize sector to 0
same_trk:
        mov bx,xltbl             ;get translation table address
        or bx,bx! jz no_trans    ;if xlt <> 0
          xlat al                ; translate sector number
no_trans:
        xor bh,bh
        mov bl,al                ;sector # is used as the index
        shl bx,1                 ; into the dma offset table
        mov ax,dmaoff
        mov dmatbl[bx],ax        ;save dma offset in table
        add ax,secsiz            ;increment dma offset by the
        mov dmaoff,ax            ; physical sector size
        inc sector               ;next sector
        dec mcnt                 ;decrement multisector count
        jnz rw_l                 ;if mcnt <> 0 store next sector dma

rw_sects:                        ;read/write sectors in dma table
;--------
        mov al,1                 ;preset error code
        xor bx,bx                ;initialize sector index
rw_sl:
        mov di,bx
        shl di,1                 ;compute index into DMA table
        cmp word ptr dmatbl[di],0ffffh
        je no_rw                 ;nop if invalid entry
          push bx! push si       ;save index and routine address
          mov ax,track           ;get track # from IOPB
          mov itrack,ax
          mov isector,bl         ;sector # is index value
          mov ax,dmatbl[di]      ;get dma offset from table
          mov idmaoff,ax
          mov ax,dmaseg          ;get dma segment from IOPB
          mov idmaseg,ax
          call si                ;call read/write routine
          pop si! pop bx         ;restore routine address and index
          or al,al! jnz err_ret  ;if error occurred return
```

Listing 5-3. (continued)

```
no_rw:
        inc bx                      ;next sector index
        cmp bx,maxsec               ;if not end of table
        jbe rw_sl                   ; go read/write next sector
err_ret:
        ret                         ;return with error code in AL
initdmatbl:     ;initialize DMA offset table
;-----------
        mov di,offset dmatbl
        mov cx,maxsec               ;length = maxsec + 1  sectors may
        inc cx                      ; index relative to 0 or 1
        mov ax,0ffffh
        push es                     ;save UDA
        push ds! pop es
        rep stosw                   ;initialize table to 0ffffh
        pop es                      ;restore UDA
        ret

;****************************************************
;*
;*      DISK I/O DATA AREA
;*
;****************************************************

zltbl   dw      0       ;translation table address
maxsec  dw      0       ;max sectors per track
secsiz  dw      0       ;sector size
dmatbl  rw      50      ;dma address table

;------------------------------------------------------------
```

Listing 5-3. (continued)

5.4   Disk Parameter Header

Each disk drive has an associated Disk Parameter Header (DPH)
that contains information about the drive and provides a scratchpad
area for certain Basic Disk Operating System (BDOS) operations.

00H	XLT	0000	00	MF	0000
08H	DPB	CSV	ALV		DIRBCB
10H	DATBCB	TBLSEG			

Figure 5-4.   Disk Parameter Header (DPH)

Table 5-4.   Disk Parameter Header Data Fields

Field	Explanation
XLT	Translation Table Address. The Translation Table Address defines a vector for logical-to-physical sector translation. If there is no sector translation (the physical and logical sector numbers are the same), set XLT to 0000h. Disk drives with identical sector skew factors can share the same translation tables. This address is not referenced by the BDOS and is only intended for use by the disk driver routines. Usually the translation table contains one byte per physical sector. If the disk has more than 256 sectors per track, the sector translation must consist of two bytes per physical sector. It is advisable, therefore, to keep the number of physical sectors per logical track to a reasonably small value to keep the translation table from becoming too large. In the case of disks with multiple heads, compute the head number from the track address rather than the sector address.
0000	Scratch Area. The 5 bytes of zeros are a scratch area which the BDOS uses to maintain various parameters associated with the drive. They must be initialized to zero by the INIT routine or the load image.

Table 5-4.   (continued)

Field	Explanation
MF	Media Flag. The BDOS resets MF to zero when the drive is logged in. The XIOS must set this flag to OFFH if it detects that the operator has opened the drive door. It must also set the global door open flag in the XIOS Header at the same time. If the flag is set to OFFH, the BDOS checks for a media change before performing the next BDOS file operation on that drive. Note that the BDOS only checks this flag when first making a system call and not during an operation. Normally, this flag is only useful in systems that support door open interrupts. If the BDOS determines that the drive contains a new disk, the BDOS logs out this drive and resets the MF field to 00H. Note: If this flag is used, removable disk performance can be optimized as if it were a permanent drive. See the description of the CKS field in the Section 5.5, "Disk Parameter Block."
DPB	Disk Parameter Block Address. The DPB field contains the address of a Disk Parameter Block that describes the characteristics of the disk drive. The Disk Parameter Block itself is described in Section 5.5. The DPB must describe the type of disk (CP/M or DOS). See IO_SELDSK in Section 5.1, and Section 5.8 for more information.
CSV	Checksum Vector Address. The Checksum Vector Address defines a scratchpad area the system uses for checksumming the directory to detect a media change. This address must be different for each Disk Parameter Header. There must be one byte for every 4 directory entries (or 128 bytes of directory). In other words, Length(CSV) = (DRM/4)+1. (DRM is a field in the Disk Parameter Block defined in Section 5.5.) If CKS in the DPB is 0000H or 8000H, no storage is reserved, and CSV may be zero. Values for DRM and CKS are calculated as part of the DPB Worksheet. If this field is initialized to OFFFFH, GENCCPM will automatically create the checksum vector and initialize the CSV field in the DPH.

Table 5-4. (continued)

Field	Explanation
ALV	Allocation Vector Address. The Allocation Vector address defines a scratchpad area which the BDOS uses to keep disk storage allocation information. This address must be different for each DPH. The Allocation Vector must contain two bits for every allocation block (one byte per 4 allocation blocks) on the disk. Or, Length(ALV) = ((DSM/8)+1)*2. The value of DSM is calculated as part of the DPB Worksheet. If the CSV field is initialized to 0FFFFH, GENCCPM automatically creates the Allocation Vector in the SYSDAT Table Area, and sets the ALV field in the DPH.
DIRBCB	Directory Buffer Control Block Header Address. This field contains the offset address of the DIRBCB Header. The Directory Buffer Control Block Header contains the directory buffer link list root for this drive. See Section 5.6, "Buffer Control Block Data Area." The BDOS uses directory buffers for all accesses of the disk directory. Several DPHs can refer to the same DIRBCB, or each DPH can reference an independent DIRBCB. If this field is 0FFFFH, GENCCPM automatically creates the DIRBCB Header, DIRBCBs, and the Directory Buffer for the drive, in the SYSDAT Table Area. GENCCPM then sets the DIRBCB field to point to the DIRBCB Header.
DATBCB	Data Buffer Control Block Header Address. This field contains the offset address of the DATBCB Header. The Data Buffer Control Block Header contains the data buffer link list root for this drive (see Section 5.6, "Buffer Control Block Data Area"). The BDOS uses data buffers to hold physical sectors so that it can block and deblock logical 128-byte records. If the physical record size of the media associated with a DPH is 128 bytes, the DATBCB field of the DPH can be set to 0000H and no data buffers are allocated. If this field is 0FFFFH, GENCCPM automatically creates the DATBCB Header and DATBCBs and allocates space for the Data Buffers in the area following the RSPs.

Table 5-4.   (continued)

Field	Explanation
TBLSEG	Table Segment. The Table Segment contains the segment address of a table used for directory hashing with CP/M disks, and as a File Allocation Table (FAT) for DOS disks. For drives that support both media, it must be large enough to hold either one. If this field is set to OFFFFH, GENCCPM will automatically create the appropriate data structures following the RSP area. The size of the table is based on the DRM (Directory Maximum) field in the DPB. For support of both media the DRM field must be set to a dummy value when GENCCPM is run to create the correct size table. See Section 5.5.1 for information on setting the DRM value. The BDOS assumes the table offset to be zero.
	Hashing is optional for CP/M disks, but the table segment must be allocated for DOS media. Thus for any drive that supports DOS disks, hashing must be specified in GENCCPM. If directory hashing is not used (CP/M media only used in this drive), set HSTBL to zero. Including a hash table dramatically improves disk performance. Each DPH using hashing must reference a unique hash table. If a hash table is desired, Length(hash_table) = 4*(DRM+1) bytes. DRM is computed as part of the DPB Worksheet. In other words, each entry in the hash table must hold four bytes for each directory entry of the disk. If this field is OFFFFH, GENCCPM will automatically create the appropriate data structures following the RSP area.
	Note: The data areas for the Data Buffers and Hash Tables are not made part of the CCPM.SYS file by GENCCPM.

Listing 5-4 illustrates the DPH definition:

```
;********************************
;*
;*      DPH Definition
;*
;********************************

xlt      equ      word ptr 0
mf       equ      byte ptr 5
dpb      equ      word ptr 8
csv      equ      word ptr 10
alv      equ      word ptr 12
dirdb    equ      word ptr 14
datbcb   equ      word ptr 16
tblseg   equ      word ptr 18

dpbase   equ      offset $          ;Base of Disk Parameter Headers

dpe0     dw       xlt0             ;Translate Table
         db       0,0,0            ;Scratch Area
         db       0                ;Media Flag
         db       0,0              ;Scratch Area
         dw       dpb0             ;Dsk Parm Block
         dw       0FFFFH,0FFFFH    ;Check, Alloc Vectors
         dw       0FFFFH           ;Dir Buff Cntrl Blk
         dw       0FFFFH           ;Data Buff Cntrl Blk
         dw       0FFFFH           ;Table Segment
;-----------------------------------------------------------------
```

Listing 5-4. DPH Definition

Given n disk drives, the DPHs can be arranged in a table whose first
row of 20 bytes corresponds to drive 0, with the last row
corresponding to drive n-1. The DPH Table has the following format:

For automatic table generation by GENCCPM,
set these fields to 0FFFFH:

DPH_TBL:									
00	XLT00	0000H	0000H	0000H	DPB00	CSV00	ALV00	DIR00	DAT00	HST00
01	XLT01	0000H	0000H	0000H	DPB01	CSV01	ALV01	DIR00	DAT00	HST01

(and so forth)

Figure 5-5. DPH Table

where the label DPH_TBL defines the offset of the DPH Table in the
XIOS.

The IO_SELDSK Function, defined in Section 5.1, returns the offset
of the DPH from the beginning of the SYSDAT segment for the selected
drive. The sequence of operations in Listing 5-5 returns the table
offset, with a 0000H returned if the selected drive does not exist.

```
;*************************************************
;*                                               *
;*              DISK IO CODE AREA                 *
;*                                               *
;*************************************************

;==========
IO_SELDSK:       ; Function 7:  Select Disk
;==========
;       entry:   CL = disk to be selected
;                DL = 00h if disk has not been previously selected
;                   = 01h if disk has been previously selected
;       exit:    AX = 0 if illegal disk
;                   = offset of DPH relative from
;                       XIOS Data Segment
```

Listing 5-5. SELDSK XIOS Function

```
        xor bx,bx              ; Get ready for error
        cmp cl,15              ; Is it a valid drive
        ja sel_ret            ; If not just exit
          mov bl,cl
          shl bx,1             ; Index into the Dph's
          mov bx,dph_tbl[bx]   ; get DPH address from table
                              ; in XIOS Header
          or dl,dl             ; First time select?
          jnz sel_ret          ; No, exit
            mov ch,0           ; Yes, set up DPH
            mov si,cx
            shl si,1
            call wordptr sel_tbl[si]
sel_ret:
        mov ax,bx
        ret

;-----------------------------------------------------------------
```

Listing 5-5. (continued)

The Translation Vectors, XLT00 through XLTn-1, whose offsets are
contained in the DPH Table as shown in Figure 5-5, are located
elsewhere in the XIOS, and correspond one-for-one with the logical
sector numbers zero through the sector count-1.

5.5 Disk Parameter Block

The Disk Parameter Block (DPB) contains parameters that define the
characteristics of each disk drive. The Disk Parameter Header (DPH)
points to a DPB thereby giving the BDOS necessary information on how
to access a disk. Several DPHs can address the same DPB if their
drive characteristics are identical.

When a drive supports both CP/M and DOS media, the IO_SELDSK routine
must determine the type of media currently in the drive and return a
DPH with a pointer to a DPB with the correct values. The standard
CP/M DPB is shown in Figure 5-6. For DOS media, the standard DPB is
extended as shown in Figure 5-7. Each field of the standard DPB is
described in Table 5-5. The extended DPB is described in Table 5-6.
A worksheet is included to help you calculate the value for each
field.

OOH	SPT		BSH	BLM	EXM	DSM		DRM...
08H	..DRM	ALO	AL1	CKS		OFF		PSH
10H	PRM							

Figure 5-6. Disk Parameter Block Format

Table 5-5. Disk Parameter Block Data Fields

Field	Explanation
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 PHM.
BSH	Allocation Block Shift Factor. This value is used by the BDOS to easily calculate a block number, given a logical record number, by shifting the record number BSH bits to the right. BSH is determined by the allocation block size chosen for the disk drive.
BLM	Allocation Block Mask. This value is used by the BDOS to easily calculate a logical record offset within a given block though masking a logical record number with BLM. The BLM is determined by the allocation block size.
EXM	Extent Mask. The Extent Mask determines the maximum number of 16K logical extents contained in a single directory entry. It is determined by the allocation block size and the number of blocks.
DSM	Disk Storage Maximum. The Disk Storage Maximum defines the total storage capacity of the disk drive. This equals the total number of allocation blocks for the drive, minus 1. 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 255.

Table 5-5.   (continued)

Field	Explanation
DRM	Directory Maximum. The Directory Maximum defines the total number of directory entries on this disk drive. This equals the total number of directory entries that can be kept in the allocation blocks reserved for the directory, minus 1. Each directory entry is 32 bytes long. The maximum number of blocks that can be allocated to the directory is 16, which determines the maximum number of directory entries allowed on the disk drive. At system generation time DRM must be set to allow enough space in TBLSEG for both the hash table and the FAT if both CP/M and DOS media can be used in the drive. See Section 5.5.1 "Disk Parameter Block Worksheet" for information on how to calculate the value for system generation.
ALO, AL1	Directory Allocation Vector. The Directory Allocation Vector is a bit map that is used to quickly initialize the first 16 bits of the Allocation Vector that is built when a disk drive is logged in. Each bit, starting with the high-order bit of ALO, represents an allocation block being used for the directory. ALO and AL1 determine the amount of disk space allocated for the directory.
CKS	Checksum Vector Size. The Checksum Vector Size determines the required length, in bytes, of the directory checksum vector addressed in the Disk Parameter Header. Each byte of the checksum vector is the checksum of 4 directory entries or 128 bytes. A checksum vector is required for removable media in order to insure the integrity of the drive. The high-order bit in the CKS field indicates a permanent drive and allows far better performance by delaying writes. Typically, hard disk systems have the value 8000H, indicating no checksumming and permanent media. On machines that can detect the door open for removable media, a special case occurs where checksumming is only done when the Media Flag (MF) byte in the DPH is set to OFFH. Normally, the disk is treated like a permanent drive, allowing more optimal use. In this case, adding 8000H to the CKS value indicated a permanent drive with checksumming.

Table 5-5.   (continued)

Field	Explanation
OFF	Track Offset. The Track Offset is the number of reserved tracks at the beginning of the disk. OFF is equal to the zero-relative track number on which the directory starts. It is through this field that more than one logical disk drive can be mapped onto a single physical drive. Each logical drive has a different Track Offset and all drives can use the same physical disk drivers.
PSH	Physical Record Shift Factor. The Physical Record Shift Factor is used by the BDOS to quickly calculate the physical record number from the logical record number. The logical record number is shifted PSH bits to the right to calculate the physical record. Note: In this context, physical record and physical sector are equivalent terms.
PRM	Physical Record Mask. The Physical Record Mask is used by the BDOS to quickly calculate the logical record offset within a physical record by masking the logical record number with the PRM value.

```
;*****************************
;*
;*    DPB Definition
;*
;*****************************

spt     equ     word ptr 0
bsh     equ     byte ptr 2
blm     equ     byte ptr 3
exm     equ     byte ptr 4
dsm     equ     word ptr 5
drm     equ     word ptr 7
al0     equ     byte ptr 9
al1     equ     byte ptr 10
cks     equ     word ptr 11
off     equ     word ptr 13
psh     equ     byte ptr 15
prm     equ     byte ptr 16
```

Listing 5-6.   DPB Definition

```
dpb0    equ     offset $        ;Disk Parameter Block
        dw      26              ;Sectors Per Track
        db      3               ;Block Shift
        db      7               ;Block Mask
        db      0               ;Extnt Mask
        dw      242             ;Disk Size - 1
        dw      63              ;Directory Max
        db      192             ;Alloc0
        db      0               ;Alloc1
        dw      16              ;Check Size
        dw      2               ;Offset
        db      0               ;Phys Sec Shift
        db      0               ;Phys Rec Mask
;-----------------------------------------------------------------
```

Listing 5-6.   (continued)

Figure 5-7 shows the extended DPB; Table 5-6 describes its fields.

00H	EXTFLAG		NFATS		NFATRECS		NCLSTRS	
08H	CLSIZE		FATADD		SPT		BSH	BLM
10H	EXM	DSM	DRM		ALO	AL1	CKS...	
18H	..CKS	OFF	PSH	PHM				

Figure 5-7.   Extended Disk Parameter Block Format

Table 5-6. Extended Disk Parameter Block Data Fields

Field	Explanation
EXTFLAG	Extended DPB Flag. The extended DPB flag is used to determine the media format currently in the drive. If EXTFLAG is set to 0FFFFH the drive contains DOS media. For CP/M media, the first field in the DPB is SPT (Sectors Per Track) and the DPB is not extended.
NFATS	Number of File Allocation Tables. This is the number of file allocation tables contained on the DOS disk. Multiple copies of the FAT can be kept on the disk as a backup if a read or write error occurs.
NFATRECS	Number of File Allocation Table Records. The number of physical sectors in the file allocation table.
NCLSTRS	Number of Clusters. The number of clusters on the DOS disk. Cluster 2 is the first data cluster to be allocated following the directory, and cluster NCLSTRS - 1 is the last available cluster on the disk.
CLSIZE	Cluster Size. The number of bytes per data cluster. This must be a multiple of the physical sector size.
FATADD	File Allocation Table Address. The physical record number of the first file allocation table on the DOS disk.
SPT	Sectors Per Track. Same as CP/M (Table 5-5).
BSH	Allocation Block Shift Factor. Same as CP/M. Used with BLM and DSM to define media capacity to CP/M. See Table 5-5.
BLM	Allocation Block Mask. See BSH.
EXM	Extent Mask. Must be zero (00H) for DOS media.
DSM	Disk Storage Maximum. See BSH.

Table 5-6.   (continued)

Field	Explanation
DRM	Directory Maximum. The number of entries - 1 in the root directory. At system generation time DRM must be set to allow enough space in TBLSEG for both the hash table and the FAT if both CP/M and DOS media can be used in the drive. See Section 5.5.1 "Disk Parameter Block Worksheet" for information on how to calculate the value for system generation.
AL0, AL1	Not used for DOS media.
CKS	Checksum Vector Size. Same as CP/M (Table 5-5).
OFF	Track Offset. Same as CP/M (Table 5-5).
PSH	Physical Record Shift Factor. Same as CP/M (Table 5-5).
PRM	Physical Record Mask. Same as CP/M (Table 5-5).

Listing 5-7 illustrates the extended DPB definition:

```
;*****************************
;*
;*    Extended DPB Definition
;*
;*****************************

extflag    equ      word ptr 0
nfats      equ      word ptr 2
nfatrecs   equ      word ptr 4
nclstrs    equ      word ptr 6
clsize     equ      word ptr 8
fatadd     equ      word ptr 10
spt        equ      word ptr 12
bsh        equ      byte ptr 14
blm        equ      byte ptr 15
exm        equ      byte ptr 16
dsm        equ      word ptr 17
drm        equ      word ptr 19
al0        equ      byte ptr 21
all        equ      byte ptr 22
cks        equ      word ptr 23
off        equ      word ptr 25
psh        equ      byte ptr 27
prm        equ      byte ptr 28

dpb0       equ      offset $      ;Disk Parameter Block
           dw       0FFFFh        ;Dos media - extended DPB
           dw       2             ;Number of FATS
           dw       6             ;Number FAT sectors
           dw       500           ;Number of clusters
           dw       1024          ;Cluster Size
           dw       1             ;Sector address of FAT
           dw       26            ;Sectors Per Track
           db       3             ;Block Shift
           db       7             ;Block Mask
           db       0             ;Extnt Mask
           dw       499           ;Disk Size - 1
           dw       67            ;Directory Max
           db       0             ;Alloc0
           db       0             ;Alloc1
           dw       17            ;Check Size
           dw       0             ;Offset
           db       0             ;Phys Sec Shift
           db       0             ;Phys Rec Mask

;----------------------------------------------------------------
```

Listing 5-7. Extended DPB Definition

5.5.1 Disk Parameter Block Worksheet

This worksheet is intended to help you create a Disk Parameter Block
containing the specifications for the particular disk hardware you
are implementing. After calculating the disk parameters according
to the directions given below, enter the value into the disk
parameter list following the Worksheet. That way, all the values
you have calculated will be in one place for a convenient reference.
The following steps, which result in values to be placed in the DPB,
are labeled "field in Disk Parameter Block".

In this worksheet, the fields common to both DPBs are calculated
first, then the fields for the extended (DOS) DPB.

<A> Allocation Block Size

    Concurrent CP/M allocates disk space in a unit known as an
allocation block. This is the minimum allocation of disk space
given to a file. This value may be 1024, 2048, 4096, 8192, or
16384 decimal bytes, or 400H, 800H, 1000H, 2000H, or 4000H
bytes, respectively. Values for DOS disks might differ from
this range. Choosing a large allocation block size allows more
efficient usage of directory space for large files and allows a
greater number of directory entries. On the other hand, a
large allocation block size increases the average wasted space
per disk file. This is the allocated disk space beyond the
logical end of a disk file. Also, choosing a smaller block
size increases the size of the allocation vectors because there
is a greater number of smaller blocks on the same size disk.
Several restrictions on the block size exist. If the block
size is 1024 bytes, there cannot be more than 255 blocks
present on a logical drive. In other words, if the disk is
larger than 256K bytes, it is necessary to use at least 2048-
byte blocks.

<B> BSH   Block Shift field in Disk Parameter Block
<C> BLM   Block Mask field in Disk Parameter Block

    Determine the values of BSH and BLM from the following table
given the value <A>.

Table 5-7. BSH and BLM Values

<A>	BSH	BLM
1,024	3	7
2,048	4	15
4,096	5	31
8,192	6	63
16,384	7	127

Note: Values for DOS disks might extend beyond this range.

<D> Total Allocation Blocks

Determine the total number of allocation blocks on the disk
drive. The total available space on the drive, in bytes, is
calculated by multiplying the total number of tracks on the
disk, minus reserved operating system tracks, by the number of
sectors per track and the physical sector size. This figure is
then divided by the allocation block size determined in <A>
above. This latter value, rounded down to the next lowest
integer value, is the Total Allocation Blocks for the drive.

<E> DSM   Disk Size Max field in Disk Parameter Block

The value of DSM equals the maximum number of allocation blocks
that this particular drive supports, minus 1.

Note: The product (Allocation Block Size)*(DSM+1) is the
total number of bytes the drive holds and must be within the
capacity of the physical disk, not counting the reserved
operating system tracks.

<F> EXM   Extent Mask field in Disk Parameter Block

For CP/M, obtain the value of EXM from the following table,
using the values of <A> and <E>. (N/A = not available). For
DOS, EXM must be zero.

<p align="center">Table 5-8. EXM Values</p>

<A>	If <E> is less than 256	If <E> is greater than or equal to 256
1,024	0	N/A
2,048	1	0
4,096	3	1
8,192	7	3
16,384	15	7

<G> Directory Blocks

Determine the number of Allocation Blocks reserved for the
directory. This value must be between 1 and 16.

<H> Directory Entries per Block

From the following table, determine the number of directory
entries per Directory Block, given the Allocation Block size,
<A>.

Table 5-9. Directory Entries per Block Size

<A>	# entries
1,024	32
2,048	64
4,096	128
8,192	256
16,384	512

<I> Total directory entries

Determine the total number of Directory Entries by multiplying
<G> by <H>.

<J> DRM   Directory Max field in Disk Parameter Block

Determine DRM by subtracting 1 from <I>. This is the value
that must be in the DRM field at run time.

The DRM field is also used by GENCCPM to allocate the hash
table for CP/M or the FAT for DOS. If both types of media are
allowed in the drive, DRM must be set to allocate the space
needed for the largest of the hash table or the FAT. The value
(I-1) calculated above will allocate the correct amount of
space for the CP/M hash table. The value to allocate space for
the FAT is calculated by:

    DRM := (NFATRECS * 2 ^ PSH * 128) / 4

The values for this equation can be found in <T>, and <P>
calculated below. Set DRM to the largest of the two values for
system generation. Set it to I - 1 at run time.

<K> AL0, AL1   Directory Allocation vector 0, 1
**   field in Disk Parameter Block**

For CP/M disks determine AL0 and AL1 from the following table,
given the number of Directory Blocks, <G>. DOS disks do not
use these fields.

Table 5-10. ALO, AL1 Values

<G>	AL0	AL1	<G>	AL0	AL1
1	80H	00H	9	0FFH	80H
2	0C0H	00H	10	0FFH	0C0H
3	0E0H	00H	11	0FFH	0E0H
4	0F0H	00H	12	0FFH	0F0H
5	0F8H	00H	13	0FFH	0F8H
6	0FCH	00H	14	0FFH	0FCH
7	0FEH	00H	15	0FFH	0FEH
8	0FFH	00H	16	0FFH	0FFH

<L> CKS   Checksum field in Disk Parameter Block

Determine the Size of the Checksum Vector. If the disk drive
media is permanent, then the value should be 8000H. If the
disk drive media is removable, the value should be ((<I>-
1)/4)+1. If the disk drive media is removable and the Media
Flag is implemented (door open can be detected through
interrupt), CKS should equal (((<I>-1)/4)+1)+ 8000H.   The
Checksum Vector should be CKS bytes long and addressed in the
DPH.

<M> OFF   Offset field in Disk Parameter Block

The OFF field determines the number of tracks that are skipped
at the beginning of the physical disk. The BDOS automatically
adds this to the value of TRACK in the IOPB and can be used as
a mechanism for skipping reserved operating system tracks, or
for partitioning a large disk into smaller logical drives.

<N> Size of Allocation Vector

In the DPH, the Allocation Vector is addressed by the ALV
field. The size of this vector is determined by the number of
Allocation Blocks. Each byte in the vector represents four
blocks, or Size of Allocation Vector = ((<E>/8)+1)*2.

<O> Physical Sector Size

Specify the Physical Sector Size of the Disk Drive. Note that
the Physical Sector Size must be greater than or equal to 128
and less than 4096 or the Allocation Block Size, whichever is
smaller. This value is typically the smallest unit that can be
read or written to the disk. This field must be filled in for
PC-MODE.

<P> PSH   Physical record SHift field in Disk Parameter Block
<Q> PRM   Physical Record Mask in Disk Parameter Block

     Determine the values of PSH and PRM from the following table
     given the Physical Sector Size. These fields must be filled in
     for PC-MODE.

                    Table 5-11. PSH and PRM Values

<O>	PSH	PRM
128	0	0
256	1	1
512	2	3
1024	3	7
2048	4	15
4096	5	31

<R> EXTFLAG   DPB Extended Flag

     If this is the DPB for a DOS disk, the DPB is an extended DPB
     and this field must be 0FFFFH.

<S> NFATS   Number of File Allocation Tables

     This field must be set to the number of file allocation tables
     on the disk currently in the drive.

<T> NFATRECS   Number of FAT Records

     This field is the number of physical sectors in the file
     allocation table. This value can be calculated from the number
     of clusters <U> and the physical sector size <O> using the
     following formula:

          <T> := (<U>* 1.5 + <O> - 1) / <O>

<U> NCLSTRS   Number of Clusters

     This field is the number of clusters on the DOS disk.

<V> CLSIZE   Cluster Size

     This field is the number of bytes per cluster. Clusters are
     similar to CP/M allocation blocks. See <A> above.

<W>   FATADD   File Allocation Table Address

This field is the physical sector number of the first file
allocation table on the DOS disk.

5.5.2 Disk Parameter List Worksheet

<A> Allocation Block Size   _____

<B> BSH field   in Disk Parameter Block   _____

<C> BLM field   in Disk Parameter Block   _____

<D> Total Allocation Blocks   _____

<E> DSM field   in Disk Parameter Block   _____

<F> EXM field   in Disk Parameter Block   _____

<G> Directory Blocks   _____

<H> Directory Entries per Block   _____

<I> Total directory entries   _____

<J> DRM field   in Disk Parameter Block   _____

<K> AL0,AL1 fields in Disk Parameter Block   _____

<L> CKS field   in Disk Parameter Block   _____

<M> OFF field   in Disk Parameter Block   _____

<N> Size of Allocation Vector   _____

<O> Physical Sector Size   _____

<P> PSH field   in Disk Parameter Block   _____

<Q> PRM field   in Disk Parameter Block   _____

<R> EXTFLAG field   in Extended Disk Parameter Block   _____

<S> NFATS field   in Extended Disk Parameter Block   _____

<T> NFATRECS field   in Extended Disk Parameter Block   _____

<U> NCLSTRS field   in Extended Disk Parameter Block   _____

<V> CLSIZE field   in Extended Disk Parameter Block   _____

<W> FATADD field   in Extended Disk Parameter Block   _____

5.6 Buffer Control Block Data Area

The Buffer Control Blocks (BCBs) locate physical record buffers for
the BDOS. BCBs are usually generated automatically by GENCCPM. The
BDOS uses the BCB to manage the physical record buffers during
processing. More than one Disk Parameter Header (DPH) can specify
the same list of BCBs. The BDOS distinguishes between two kinds of
BCBs, directory buffers, referenced by the DIRBCB field of the DPH,
and data buffers, referenced by DATBCB field of the DPH.

The DIRBCB and DATBCB fields each contain the offset address of a
Buffer Control Block Header. The BCB Header contains the offset of
the first BCB in a linked list of BCBs. Each BCB has a LINK field
containing the address of the next BCB in the list, or 0000H if it
is the last BCB. All BCB Headers and BCBs must reside within the
SYSDAT segment.

BCBLR	MBCBP
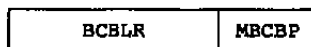
Figure 5-8. Buffer Control Block Header

Table 5-12. Buffer Control Block Header Data Fields

Field	Explanation
BCBLR	Buffer Control Block List Root. The Buffer Control Block *List Root* points to the first BCB in a linked list of BCB's.
MBCBP	Maximum BCB's per Process. The MBCBP is the maximum number of BCB's that the BDOS can allocate to any single process at one time. If the number of BCB's required by a process is greater than MBCBP, the BDOS reuses BCB's previously allocated to this process on a least-recently-used (LRU) basis.

Listing 5-8 illustrates the BCB Header definition:

```
;****************************
;*
;*    BCB Header Definition
;*
;****************************

bcblr    equ      word ptr 0
mbcbp    equ      byte ptr 2


dirbcb   dw       dirbcb0         ;BCB List Head
         db       4               ;Max # BCB's/Process

;-----------------------------------------------------------------
```

Listing 5-8.   BCB Header Definition

Figure 5-9 shows the format of the Directory Buffer Control Block:

00H:	DRV	RECORD	WFLG	SEQ	TRACK
08H:	SECTOR	BUFOFF	LINK	PDADR	
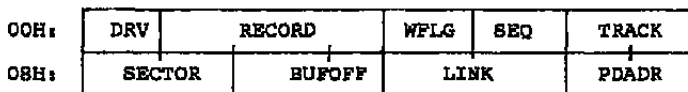
Figure 5-9. Directory Buffer Control Block (DIRBCB)

Table 5-13. DIRBCB Data Fields

Field	Explanation
DRV	Logical Drive Number. The Logical Drive Number identifies the disk drive associated with the physical sector contained in the buffer. The initial value of the DRV field must be OFFH. If DRV = OFFh then the BDOS considers that the buffer contains no data and is available for use.
RECORD	Record Number. The Record Number identifies the logical record position of the current buffer for the specified drive. The record number is relative to the beginning of the logical disk, where the first record of the directory is logical record number zero.
WFLG	Write Pending Flag. The BDOS sets the Write Pending Flag to OFFH to indicate that the buffer contains unwritten data. When the data are written to the disk, the BDOS sets the WFLG to zero to indicate that the buffer is no longer dirty.
SEQ	Sequential Access Counter. The BDOS uses the Sequential Access Counter during blocking and deblocking to detect whether the buffer is being accessed sequentially or randomly. If sequential access is used, the BDOS allows reuse of the buffer to avoid consumption of all buffers during sequential I/O.
TRACK	Logical Track Number. The TRACK is the logical track number for the current buffer.
SECTOR	Physical Sector Number. SECTOR is the logical sector number for the current buffer.
BUFOFF	Buffer Offset. For DIRBCBs, this field equals the offset address of the buffer within SYSDAT.
LINK	Link to next DIRBCB. The Link field contains the offset address of the next BCB in the linked list, or 0000H, if this is the last BCB in the linked list.
PDADR	Process Descriptor Address. The BDOS uses the Process Descriptor Address to identify the process which owns the current buffer.

The buffer associated with the BCB must be large enough to accommodate the largest physical record (equivalent to physical sector) associated with any DPH referencing the BCBs. The initial value of the DRV field must be 0FFH. When the DRV field contains 0FFH, the BDOS considers that the buffer contains no data and is available for use. When WFLG equals 0FFH, the buffer contains data that the BDOS has to write to the disk before the buffer is available for other data.

Directory BCBs never have the BCB WFLG parameter set to 0FFH because directory buffers are always written immediately. The BDOS postpones only data buffer write operations. Thus, only data BCBs can have dirty buffers.

The data and directory BCBs must be separate. This is to ensure that a buffer with a clear WFLG is available when the BDOS verifies the directory. If all the buffers contain new data (WFLG set to 0FFH), the BDOS has to perform a write before it can verify that the disk media has changed. This could result in data being written on the wrong disk inadvertently. The following listing illustrates the DIRBCB definition:

```
;********************************
;*
;*    DIRBCB Definition
;*
;********************************

drv       equ      byte ptr 0
record    equ      byte ptr 1
wflg      equ      byte ptr 4
seq       equ      byte ptr 5
track     equ      word ptr 6
sector    equ      word ptr 8
bufoff    equ      word ptr 10
link      equ      word ptr 12
pdadr     equ      word ptr 14


dirbcb0 db        0ffh                  ;Drive
        rb        3                     ;Record
        rb        2                     ;Pending, Sequence
        rw        2                     ;Track, Sector
        dw        dirbuf0               ;Buffer Offset
        dw        dirbcb1               ;Link
        rw        1                     ;PD Address

;------------------------------------------------------------------
```

Listing 5-9. DIRBCB Definition

Figure 5-10 shows the format of the Data Buffer Control Block
(DATBCB):

OOH:	DRV	RECORD	WFLG	SEQ	TRACK
O8H:	SECTOR	BUFSEG	LINK		PDADR

Figure 5-10. Data Buffer Control Block (DATBCB)

The DATBCB is identical to the DIRBCB, except for the BUFSEG Field
described in Table 5-14.

Table 5-14. DATBCB Data Fields

Field	Explanation
BUFSEG	Buffer Segment. For BCBs describing data buffers, this field equals the segment address of the Data Buffer. The offset address of the buffer is assumed to be zero. The actual buffer can be anywhere in memory on a paragraph boundary that is not in the system TPA.

Listing 5-10 illustrates the DATBCB definition:

```
;*******************************
;*
;*    DATBCB Definition
;*
;*******************************

drv      equ       byte ptr 0
record   equ       byte ptr 1
wflg     equ       byte ptr 4
seq      equ       byte ptr 5
track    equ       word ptr 6
sector   equ       word ptr 8
bufseg   equ       word ptr 10
link     equ       word ptr 12
pdadr    equ       word ptr 14

datbcb0 db         0ffh              ;Drive
        rb         3                 ;Record
        rb         2                 ;Pending, Sequence
        rw         2                 ;Track, Sector
        dw         dirbuf0           ;Buffer Segment
        dw         dirbcb1           ;Link
        rw         1                 ;PD Address

;------------------------------------------------------------
```

Listing 5-10.   DATBCB Definition

5.7 Memory Disk Application

A memory disk or M disk is a prime example of the ability of the
Basic Disk Operating System to interface to a wide variety of disk
drives.   A memory disk uses an area of RAM to simulate a small
capacity disk drive, making a very fast temporary disk. The M disk
can be specified by GENCCPM as the temporary drive. The example
XIOS implements an M disk for the IBM PC. This section discusses a
similar M disk implementation as shown in Listing 5-11.

In Listing 5-11, the M disk memory space begins at the 0C000H
paragraph boundary and extends for 128 Kbytes, through the 0DFFFH
paragraph.   It is assumed the XIOS INIT routine calls the
INIT_M_DSK: code, which initializes the directory area of the M
disk, the first 16 Kbytes, to 0E5H.

Both the M disk READ and WRITE routines first call the MDISK_CALC:
routine.   This code calculates the paragraph address of the current
sector in memory, and the number of words of data to read or write.
The number of sectors per track for the M disk is set to 8,
simplifying the calculation of the sector address to a simple shift-
and-add operation.   The multisector count is multiplied by the
length of a sector to give the number of words to transfer.

The READ_M_DISK: routine gets the current DMA address from the IOPB
on the stack, and using the parameters returned by the MDISK_CALC:
routine, block-moves the requested data to the DMA buffer.   The
WRITE_M_DISK: routine is similar except for the direction of data
transfer.

A Disk Parameter Block for the M disk, illustrated at the end of the
example, is provided for reference. A hash table is provided in
order to increase performance to the maximum. However, this field
can be set to zero if directory hashing is not desirable due to
space limitations.

Listing 5-11 illustrates an M disk implementation:

```
;****************************************************
;       M DISK EQUATES
;****************************************************

mdiskbase        equ     0C000h    ;base paragraph
                                   ;address of mdisk

;****************************************************
;       M DISK INITIALIZATION
;****************************************************
init_m_dsk:
        mov cx,mdiskbase
        push es | mov es,cx
        xor di,di
        mov ax,0e5e5h             ;check if already initialized
        cmp es:[di],ax | je mdisk_end
          mov cx,2000h            ;Initialize 16K bytes
          rep stos ax            ;of M disk directory to 0E5h's
mdisk_end:
        pop es
        ret

;****************************************************
;       M DISK CODE
;****************************************************

;=======
IO_READ:        ; Function 11: Read sector
;=======
; Reads the sector on the current disk, track and
; sector into the current DMA buffer.
;       entry:  parameters on stack
;       exit:   AL = 00 if no error occurred
;               AL = 01 if an error occurred

read_m_dsk:
;-----------
        call mdisk_calc              ;calculate byte address
        push es                      ;save UDA
        les di,dword ptr dmaoff      ;load destination DMA address
        xor si,si                    ;setup source DMA address
        push ds                      ;save current DS
        mov ds,bx                    ;load pointer to sector in memory
        rep movsw                    ;execute move of 128 bytes....
        pop ds                       ;then restore user DS register
        pop es                       ;restore UDA
        xor ax,ax                    ;return with good return code
        ret
```

Listing 5-11. Example M disk implementation

```
; ========
IO_WRITE:              ; Function 12: Write disk
; ========
; Write the sector in the current Dma buffer
; to the current disk on the current
; track in the current sector.
;       entry:  CL = 0 - Deferred Writes
;                    1 - nondeferred writes
;                    2 - def-wrt 1st sect unalloc blk
;       exit:   AL = 00H if no error occurred
;                  = 01H if error occurred
;                  = 02H if read only disk

write_m_dsk:
;-----------
        call mdisk_calc            ;calculate byte address
        push es                    ;save UDA
        mov es,bx                  ;setup destination DMA address
        xor di,di
        push ds                    ;save user segment register
        lds si,dword ptr dmaoff    ;load source DMA address
        rep movsw                  ;move from user to disk in memory
        pop ds                     ;restore user segment pointer
        pop es                     ;restore UDA
        xor ax,ax                  ;return no error
        ret

mdisk_calc:
;----------
;       entry:  IOPB variables on the stack
;       exit:   BX = sector paragraph address
;               CX = length in words to transfer

        mov bx,track               ;pickup track number
        mov cl,3                   ;times eight for relative
                                   ;          sector number
        shl bx,cl
        mov cx,sector              ;plus sector
        add bx,cx                  ;gives relative sector number
        mov cl,3                   ;times eight for paragraph
                                   ;          of sector start
        shl bx,cl
        add bx,mdiskbase           ;plus base address of disk
                                   ;          in memory
        mov cx,64                  ;length in words for move
                                   ;          of 1 sector
        mov al,mcnt
        xor ah,ah
        mul cx                     ;length * multisector count
        mov cx,ax
        cld
        ret
```

Listing 5-11. (continued)

```
;*********************************************************
;       M DISK - DISK PARAMETER BLOCK
;*********************************************************

dpb0    equ     offset $        ;Disk Parameter Block
        dw      8               ;Sectors Per Track
        db      3               ;Block Shift
        db      7               ;Block Mask
        db      0               ;Extnt Mask
        dw      126             ;Disk Size - 1
        dw      31              ;Directory Max
        db      128             ;Alloc0
        db      0               ;Alloc1
        dw      0               ;Check Size
        dw      0               ;Offset
        db      0               ;Phys Sec Shift
        db      0               ;Phys Sec Mask

xlt5    equ     0               ;No Translate Table
als5    equ     16*2            ;Allocation Vector Size
css5    equ     0               ;Check Vector Size
hss5    equ     (32 * 4)        ;Hash Table Size

;--------------------------------------------------------
```

Listing 5-11.   (continued)

5.8   Multiple Media Support

Disk access is controled by a number of data structures, that
describe various parameters of the disk. Some of these parameters
are set in the code of the XIOS, others are filled in by GENCCPM.
When a particular disk drive can have more than one type of disk in
it (for example different densities or CP/M and PC-DOS disks) some
of these parameters must be set at run time. This section explains
how these parameters are set up, and which ones must be changed at
run time.

Each disk drive is described by a disk parameter header (DPH) that
gives addresses for several data structures needed in using the
disk, including the Disk Parameter Block (DPB). The DPB describes
the disk in more detail, such as the size of the directory and the
total storage capacity of the drive. The information in the DPB
will be different if a different density or format disk is used.

The DPH is located by the DPH(A) through DPH(P) pointers in the XIOS
header. See Section 3.1 "XIOS Header" for more information on these
pointers. The fields in the DPH can be filled in by hard coding the
values in the XIOS or if they are set to OFFFFH, GENCCPM will
calculate and fill in the values. GENCCPM also allocates space for
the needed buffers and vectors.

If a drive supports more than one type of media, the buffers
allocated must be large enough to hold the information needed for
any of the possible media. This may require creating a dummy DPH
and DPB for GENCCPM to use while allocating the buffers. For DOS
and CP/M disks, the same table area (pointed to by TBLSEG in the
DPH) is used for the hash table (CP/M) and the FAT (DOS). The space
GENCCPM allocates for this is based on the DRM value in the DPB.
See Section 5.5.1 for information on setting DRM.

Auto Density Support is the ability to support different types of
media on the same drive. Some floppy disk drives can read many
different disk formats. Auto Density Support enables the XIOS to
determine the density of the diskette when the IO_SELDSK function is
called, and to detect a change in density when the IO_READ or
IO_WRITE functions are called.

To implement Auto Density Support or support for both CP/M and DOS
media, the XIOS disk driver must include a DPB for each disk format
expected, or routines to generate proper DPB values automatically in
real time. It must also be able to determine the type and format of
the disk when the IO_SELDSK function is called for the first time,
set the DPH to address the DPB that describes the media, and return
the address of the DPH to the BDOS. If unable to determine the
format, the IO_SELDSK function can return a zero, indicating that
the select operation was not successful. On all subsequent
IO_SELDSK calls, the XIOS must continue to return the address of the
same DPH; a return value of zero is only allowed on the initial
IO_SELDSK call.

Once the IO_SELDSK routine has determined the format of the disk,
the IO_READ and IO_WRITE routines assume this format is correct
until an error is detected. If an XIOS function encounters an error
and determines that the media has been changed to another format, it
must abandon the operation and return OFFH to the BDOS. This
prompts the BDOS to make another initial IO_SELDSK call to
reestablish the media type. XIOS routines must not modify the
drive's DPH or DPB until the IO_SELDSK call is made. This is
because the BDOS can also determine that the media has changed, and
can make an initial IO_SELDSK call even though the XIOS routines
have not detected any change.

End of Section 5

Section 6
PC-MODE Character I/O

This section describes functions that must be implemented in the
XIOS to support PC-MODE. These functions emulate some of the PC
interrupts, allowing DOS programs to run.

There are seven functions that must be added to the XIOS to support
PC-MODE. These are functions 30 through 36. This chapter describes
functions 30 through 34, that are used for character I/O. Functions
35 and 36 are for disk I/O, and are described in Section 5. Note
that the XIOS function table must be extended for these functions.
See Section 3.3 "XIOS ENTRY" for more information on the function
table.

Implementing these functions requires data structures similar to
those used in screen buffering. See Section 4.2 "Console I/O
Functions" for more information on screen buffering. Screen
buffering is assumed in the descriptions of all the routines in this
chapter.

6.1 Screen I/O Functions

Function 30, IO_SCREEN either returns the current screen mode, or
sets the screen to a certain mode. The mode tells whether the
screen is displaying text or graphics, and the screen size.
Function 31, IO_VIDEO, provides functions for getting and setting
the cursor position and attributes, as well as scrolling the screen
and writing characters. This function emulates 8 of the 16
subfunctions of DOS's interrupt 10.

```
┌──────────────────────────────────────────────────────────┐
│                                                            │
│           IO_SCREEN     GET/SET SCREEN                     │
│                                                            │
├──────────────────────────────────────────────────────────┤
│                                                            │
│          Get or Set the Current Screen                     │
│                                                            │
├──────────────────────────────────────────────────────────┤
│                                                            │
│   Entry Parameters:                                        │
│       Register AL:   1EH (30)                              │
│                CH:   0 = Set, 1 = Get                      │
│                CL:   Mode if CH = 0 (Set)                  │
│                DL:   Virtual console number                │
│                                                            │
│   Returned   Value:                                        │
│       Register AX:   Mode if CH = 1 (Get)                  │
│                AX:   FFFFH if mode not supported           │
│                      (Set)                                 │
│                      FFFEH if bad parameters               │
│                      (Set)                                 │
│                      0000H if successful (Set)             │
│                ES, DS, SS, SP  preserved                   │
│                                                            │
└──────────────────────────────────────────────────────────┘
```

IO_SCREEN can be called to either return the current screen mode
(Get) or to set the screen to a certain mode (Set). Set is
indicated by a zero in CH, Get is indicated by a 1 in CH. IO_SCREEN
is called to operate on a virtual console, indicated by DL. The
sample XIOS's keep a record of the mode of each virtual console in
the screen structure. The screen mode must be initialized to a
nonzero value when the system is initialized. This function is also
used for GSX support. See Appendix B.

When IO_SCREEN is called to set the screen mode (CH = 0), CL
contains the mode in the following format:

```
        CH              CL
    ┌──────────┬──────┬──────┐
    │   00H    │  x   │  y   │
    └──────────┴──────┴──────┘
```

where y indicates the alphanumeric modes and x indicates graphics
modes. Either x or y will have a value, the other will be zero.
The alphanumeric modes (values for y) are shown in Table 6-1. The
graphics modes (values for x) are shown in Table 6-2. The value 1
(general alphanumeric or general graphic mode) comes from the GSX
graphics system's GIOS to indicate a mode switch. The GIOS does its
own hardware initialization.

If the calling process is in the background and wants to set its
mode to graphics, IO_SCREEN must flagwait the process. The
corresponding flagset takes place in the IO_SWITCH routine, when the
process's virtual console is switched to the foreground. For
further information on the IO_SWITCH routine, see Section 4.2
"Console I/O Functions".

Set should initialize the hardware if necessary.

When IO_SCREEN is called with CH = 1 (get) it returns the screen
mode (from the screen structure) in the following format:

CH	CL	
# Cols	x	y

where # Cols is the number of columns on the screen, x is the
graphics mode (Table 6-2), and y is the alphanumeric mode (Table 6-1).

Table 6-1. Alphanumeric Modes

Y Value	Meaning
1	General alphanumeric mode
2	40 x 25 monochrome
3	40 x 25 color
4	80 x 25 monochrome
5	80 x 25 color
6 - 8	Reserved
9	80 x 25 monochrome card
10 - 15	Reserved

Table 6-2. Graphics Modes

X Value	Meaning
1	General graphics mode
2	320 x 200 color
3	320 x 200 monochrome
4	640 x 200 monochrome
5 - 15	Reserved

IO_VIDEO (Function 31) emulates 8 of the 16 subfunctions of DOS's interrupt 10. It will set and read the cursor position, scroll the screen, set and read attributes, and write characters to the screen.

```
┌─────────────────────────────────────────────────────┐
│                                                     │
│     IO_VIDEO     VIDEO INPUT/OUTPUT                  │
│                                                     │
├─────────────────────────────────────────────────────┤
│                                                     │
│            Manipulate the Video Screen              │
│                                                     │
├─────────────────────────────────────────────────────┤
│   Entry Parameters:                                 │
│        Register AL:   1FH  (31)                     │
│                 BL:   Sub Function                  │
│                 CX:   Input parameter               │
│                       (see below)                   │
│                 DX:   Input parameter               │
│                       (see below)                   │
│                                                     │
│   Returned   Value:                                 │
│        Depends on subfunction.   See below.         │
│                 ES, DS, SS, SP   preserved          │
│                                                     │
└─────────────────────────────────────────────────────┘
```

The IO_VIDEO function must implement at least 8 of the 16 subfunctions of DOS's interrupt 10. All 16 can be implemented if desired, and if the hardware supports them. The 8 required subfunctions are described below.

SET CURSOR POSITION (BL = 2)

```
        entry:  CH = row
                CL = column
                DL = virtual console number
        exit:   none
```

This function sets the cursor position to the specified row and column. It updates the cursor position in the screen structure for the specified virtual console. It also updates the physical screen if this virtual console is in the foreground.

READ CURSOR POSITION (BL = 3)

```
entry:  DL = virtual console number
exit:   AH = row
        AL = column
```

This function returns the current cursor position for the virtual
console from the screen structure.

SCROLL UP (BL = 6)

```
entry:  CX = segment of parameter structure
        DX = offset of parameter structure
exit:   none
```

This function accesses the parameter structure and scrolls up the
specified window on the virtual console. The window is specified by
giving the row and column of the upper left and lower right corners
of the rectangle. If the number of lines to scroll is 0, the window
should be cleared. The parameter structure is as follows:

0:	A	
2:	B	RSVD
4:	(row) C	(col)
6:	(row) D	(col)
8:	VC	

```
where:  A  = number of lines
        B  = attribute of blank lines
        C  = row, column of upper left
        D  = row, column of lower right
        VC = virtual console number
```

If screen buffering is implemented, scrolling must take place in the
screen buffer. If the virtual console is in the foreground, and the
physical console is a serial terminal, the display must also be
updated. Parameter B contains the attributes desired for the new
blank lines to be added in the window. The method of displaying the
scrolled window on the physical console depends on the hardware.

SCROLL DOWN (BL = 7)

```
                    entry:  CX = segment of parameter structure
                            DX = offset of parameter structure
                    exit:   none
```

This function accesses the parameter structure and scrolls down the
specified window on the virtual console, similar to the previous
subfunction. The parameter structure is as follows:

```
0:  ┌─────────────────────┐
    │          A          │
2:  ├──────────┬──────────┤
    │    B     │   RSVD   │
4:  ├──────────┴──────────┤
    │  (row)  C  (col)    │
6:  ├─────────────────────┤
    │  (row)  D  (col)    │
8:  ├──────────┬──────────┘
    │    VC    │
    └──────────┘
```

```
            where:  A  = number of lines
                    B  = attribute of blank lines
                    C  = row, column of upper left
                    D  = row, column of lower right
                    VC = virtual console number
```

Refer to scroll up above for more information.

READ ATTRIBUTE/CHARACTER (BL = 9)

```
                    entry:  DL = virtual console number
                    exit:   AH = attribute
                            AL = character
```

This function accesses the screen structure for the virtual console
and returns the character and the attribute byte for the current
cursor position.

In the example XIOS's, this subfunction involves: 1) Using the
virtual console number to look up the screen structure. 2) Get the
screen buffer address and cursor position from the screen structure.
3) Look up the screen buffer, and use the cursor position as an
offset to get the current character and attribute byte.

WRITE ATTRIBUTE/CHARACTER (BL = 9)

```
entry:  CX = segment of parameter structure
        DX = offset of parameter structure
exit:   none
```

This function writes a character and an attribute byte to a screen image. The new character and attribute are written at the current cursor position, and the cursor position moved to the new character. This may involve handling an end of line or end of screen condition. Any number of the same character and attributes can be written by specifying the count in CX. If this virtual console is in the foreground, and the physical console is a serial terminal, it must be updated with the new characters and attributes. The parameter structure is as follows:

0:	RSVD	A
2:	RSVD	B
4:	C	
6:	RESERVED	
8:	VC	

```
where:  A = character
        B = attributes
        C = number of characters to repeat
        VC = virtual console number
```

WRITE CHARACTER (BL = 10)

```
entry:  CX = segment of parameter structure
        DX = Offset of parameter structure
exit:   none
```

This function writes a character to the screen buffer at the current cursor position, with the same attribute(s) as the previous character. The character can be repeated by specifying a count in C. If the virtual console is in the foreground, and the physical console is a serial terminal, it must also be updated. The parameter structure is as follows:

0:	RSVD	A
2:	RESERVED	
4:	C	
6:	RESERVED	
8:	VC	

where: A = character
        C = number of characters to repeat
        VC = virtual console number

WRITE SERIAL CHARACTER (DL = 14)

        entry: CL = character
                DL = virtual console number
        exit: none

This function writes a character to the screen image at the current
cursor position, and to the physical screen if the virtual console
is in the foreground. It functions similarly to write character
(above) but does not allow repeated characters. This is a teletype
write, and does not allow escape sequences.

6.2 Keyboard Functions

These two functions are used for handling function keys and the
shift status of the keyboard when running in PC-MODE.

```
┌─────────────────────────────────────────────────────────┐
│                                                         │
│            IO_KEYBD    KEYBOARD MODE                     │
│                                                         │
├─────────────────────────────────────────────────────────┤
│                                                         │
│            Enable/Disable PC-MODE                        │
│                                                         │
├─────────────────────────────────────────────────────────┤
│                                                         │
│      Entry Parameters:                                   │
│           Register AL:   20H (32)                        │
│                    CL:   1 = Enable                      │
│                          2 = Disable                     │
│                    DL:   Virtual Console Number          │
│                                                         │
│      Returned    Value:                                  │
│           Register AX:   0 if OK                         │
│                          FFFFH if error                  │
│                    ES, DS, SS, SP  preserved             │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

IO_KEYBD is a signal to tell whether PC-MODE is active or not. When
it is enabled, the console is running a PC program, and several
functions must behave differently. These differences have to do
with the function keys on the keyboard, and the 25th line on the
screen.

Enabling or disabling IO_KEYBD tells IO_CONIN (See Section 4.2)
whether to pass function keys to the caller or not. Normally
(disabled) all function keys not used by the XIOS (those that do not
have an associated function, such as screen switch) are ignored on
input. If IO_KEYBD is enabled, IO_CONIN must pass all 16 bit
function key codes to the caller. See Section 6.4.

Many PC applications use the 25th line of the display. Thus when
you are in PC-MODE, IO_STATLINE must not display. See section 4.2
for more information on IO_STATLINE.

This variable can also be used in the XIOS for any other functions
that need to know if a console is in PC-MODE. For example, it could
be used to indicate if 24 or 25 lines need to be buffered.

```
+-------------------------------------------------------+
|                                                       |
|            IO_SHFT     SHIFT STATUS                    |
|                                                       |
+-------------------------------------------------------+
|                                                       |
|                 Return Shift Status                   |
|                                                       |
+-------------------------------------------------------+
|                                                       |
|   Entry Parameters:                                   |
|         Register AL:   21B (33)                        |
|                  DL:   Virtual Console Number          |
|                                                       |
|   Returned   Value:                                   |
|         Register AL:   Shift Status                    |
|                   ES, DS, SS, SP  preserved            |
|                                                       |
+-------------------------------------------------------+
```

IO_SHFT emulates PC interrupt 16 subfunction 2. It returns a bit
map showing the status of certain keys on the keyboard. The bit map
is shown in Table 6-3.

Table 6-3. Keyboard Shift Status

Bit	Meaning
7	Insert state is active
6	Caps lock state has been toggled
5	Num lock state has been toggled
4	Scroll lock state has been toggled
3	Alternate shift key depressed
2	Control shift key depressed
1	Left shift key depressed
0	Right shift key depressed

6.3   Equipment Check

```
┌─────────────────────────────────────────────────────────┐
│                                                         │
│           IO_EQCK      EQUIPMENT CHECK                  │
│                                                         │
├─────────────────────────────────────────────────────────┤
│                                                         │
│              Return Equipment Status                    │
│                                                         │
├─────────────────────────────────────────────────────────┤
│      Entry Parameters:                                  │
│           Register AL:   22H (34)                       │
│                                                         │
│      Returned   Value:                                  │
│           Register AX:   DOS bit map (Table 6-3)        │
│                          ES, DS, SS, SP   preserved     │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

IO_EQCK emulates DOS's interrupt 11. It returns a subset of DOS's
standard bit map that describes the state of the equipment. This
bit map is shown in Table 6-3.

Table 6-4.   DOS Equipment Status Bit Map

Bit	Meaning
14, 15	Number of printers attached
13	Not used
12	Game I/O attached
11 - 9	Number of RS232 cards attached
8	Not used
7, 6	Number of floppy disk drives
5, 4	Initial video mode
3, 2	Planar RAM size
1	Not used
0	IPL from floppy

6.4   PC-MODE IO_CONIN

When a virtual console is in PC-MODE (See IO_KEYBD in Section 6.2)
IO_CONIN must return extended codes for certain function keys. Most
characters are returned as their ASCII code in AL, and their scan
code in AH. The scan codes for all keys are shown in Table 6-5.
Extended keys are returned as a nul (00H) in AL and an extended code
in AH. The extended keys and the value to be returned in AH are
shown in Table 6-6.

Table 6-5.   Keyboard Scan Codes

Key	Scan Code	Key	Scan Code
A	30	Esc	1
B	48	Ctrl	29
C	46	Shift (left)	42
D	32	Shift (right)	54
E	18	Alt	56
F	33	Caps Lock	58
G	34	Num Lock	69
H	35	Scroll Lock	70
I	23	Return	28
J	36	Tab	15
K	37	backspace	14
L	38		
M	39	Numeric Keypad:	
N	49		
O	24	Home (7)	71
P	25	cursor up (8)	72
Q	16	Pg Up (9)	73
R	19	cursor left (4)	75
S	31	(5)	76
T	20	cursor right (6)	77
U	22	End (1)	79
V	47	cursor down (2)	80
W	17	PgDn (3)	81
X	45	Ins (0)	82
Y	21	Del (.)	83
Z	44	* (PrtSc)	55
1 (!)	2	-	74
2 (@)	3	+	78
3 (#)	4		
4 ($)	5	Function Keys:	
5 (%)	6		
6 (^)	7	F1	59
7 (&)	8	F2	60
8 (*)	9	F3	61
9 (()	10	F4	62
0 ())	11	F5	63
- (_)	12	F6	64
= (+)	13	F7	65
[({)	26	F8	66
] (})	27	F9	67
; (:)	39	F10	68
' (")	40		
` (~)	41		
, (<)	51		
. (>)	52		
/ (?)	53		
\ (|)	54		

Table 6-6.   Extended Keyboard Codes

Character	AH	Function
ctrl 3	3	Nul character
|<—	15	Reverse tab
Ins	82	Insert
Del	83	Delete
|	72	Cursor up
<—	75	Cursor left
—>	77	Cursor right
|	80	Cursor down
home	71	Cursor home
ctrl home	119	Control home
ctrl <—	115	Reverse word
ctrl —>	116	Advance word
Pg Dn	81	Page down
ctrl Pg Dn	118	Contrl page down
Pg Up	73	Page up
ctrl Pg Up	132	Control page up
End	79	End
ctrl End	117	Control end
ctrl PrtSc	114	Print screen
F1	59	Function key F1
F2	60	Function key F2
F3	61	Function key F3
F4	62	Function key F4
F5	63	Function key F5
F6	64	Function key F6
F7	65	Function key F7
F8	66	Function key F8
F9	67	Function key F9
F10	68	Function key F10
shift F1	84	Function key F11
shift F2	85	Function key F12
shift F3	86	Function key F13
shift F4	87	Function key F14
shift F5	88	Function key F15
shift F6	89	Function key F16
shift F7	90	Function key F17
shift F8	91	Function key F18
shift F9	92	Function key F19
shift F10	93	Function key F20

Table 6-6.   (continued)

Character	AH	Function
ctrl F1	94	Function key F21
ctrl F2	95	Function key F22
ctrl F3	96	Function key F23
ctrl F4	97	Function key F24
ctrl F5	98	Function key F25
ctrl F6	99	Function key F26
ctrl F7	100	Function key F27
ctrl F8	101	Function key F28
ctrl F9	102	Function key F29
ctrl F10	103	Function key F30
alt F1	104	Function key F31
alt F2	105	Function key F32
alt F3	106	Function key F33
alt F4	107	Function key F34
alt F5	108	Function key F35
alt F6	109	Function key F36
alt F7	110	Function key F37
alt F8	111	Function key F38
alt F9	112	Function key F39
alt F10	113	Function key F40
alt A	30	Alt A
alt B	48	Alt B
alt C	46	Alt C
alt D	32	Alt D
alt E	18	Alt E
alt F	33	Alt F
alt G	34	Alt G
alt H	35	Alt H
alt I	23	Alt I
alt J	36	Alt J
alt K	37	Alt K
alt L	38	Alt L
alt M	50	Alt M
alt N	49	Alt N
alt O	24	Alt O
alt P	25	Alt P
alt Q	16	Alt Q
alt R	19	Alt R
alt S	31	Alt S
alt T	20	Alt T
alt U	22	Alt U
alt V	47	Alt V
alt W	17	Alt W
alt X	45	Alt X
alt Y	21	Alt Y
alt Z	44	Alt Z

Table 6-6.   (continued)

Character	AH	Function
alt 1	120	Alt 1
alt 2	121	Alt 2
alt 3	122	Alt 3
alt 4	123	Alt 4
alt 5	124	Alt 5
alt 6	125	Alt 6
alt 7	126	Alt 7
alt 8	127	Alt 8
alt 9	128	Alt 9
alt 0	129	Alt 0
alt -	130	Alt -
alt =	131	Alt =

End of Section 6

Section 7
XIOS Tick Interrupt Routine

The XIOS must continually perform two DEV_SETFLAG system calls.
Once every system tick the system tick flag must be set if the TICK
Boolean in the XIOS Header is OFFH. Once every second, the second
flag must be set. This requires the XIOS to contain an interrupt-
driven tick routine that uses a hardware timer to count the time
intervals between successive system ticks and seconds.

The recommended tick unit is a period of 16.67 milliseconds,
corresponding to a frequency of 60 Hz. When operating on 50 Hz
power, use a 20-millisecond period. The system tick frequency
determines the dispatch rate for compute-bound processes. If the
frequency is too high, an excessive number of dispatches occurs,
creating a significant amount of additional system overhead. If the
frequency is too low, compute-bound processes monopolize the CPU
resource for longer periods.

Concurrent CP/M uses Flag #2 to maintain the system time and day in
the TOD structure in SYSDAT. The CLOCK process performs a
DEV_WAITFLAG system call on Flag #2, and thus wakes up once per
second to update the TOD structure. The CLOCK process also calls
the IO_STATLINE XIOS function to update the status line once per
second. If the system has more than one physical console, one
physical console is updated each second. Thus if four physical
consoles are connected, each one will be updated once every four
seconds.

The CLOCK process is an RSP and the source code is distributed in
the OEM kit. Any functions needing to be performed on a per-second
basis can simply be added to the CLOCK.RSP.

After performing the DEV_SETFLAG calls described above, the XIOS
TICK Interrupt routine must perform a Jump Far to the dispatcher
entry point. This forces a dispatch to occur and is the mechanism
by which Concurrent CP/M effects process dispatching. The double-
word pointer to the dispatcher entry used by the TICK interrupt is
located at 0038H in the SYSDAT DATA. Please see Section 3.6,
"Interrupt Devices," for more information on writing XIOS interrupt
routines.

<center>End of Section 7</center>

<center>7-1</center>

Section 8
Debugging the XIOS

This section suggests a method of debugging Concurrent CP/M, requiring CP/M-86 running on the target machine, and a remote console. Hardware-dependent debugging techniques (ROM monitor, in-circuit emulator) available to the XIOS implementor can certainly be used but are not described in this manual.

Implement the first cut of the XIOS using all polled I/O devices, all interrupts disabled including the system TICK, and Interrupt Vectors 1, 3, and 225, which are used by DDT-86 and SID-86, uninitialized. Once the XIOS functions are implemented as polling devices, change them to interrupt-driven I/O devices and test them one at a time. The TICK interrupt routine is usually the last XIOS routine to be implemented.

The initial system can run without a TICK interrupt, but has no way of forcing CPU-bound tasks to dispatch. However, without the TICK interrupt, console and disk I/O routines are much easier to debug. In fact, if other problems are encountered after the TICK interrupt is implemented, it is often helpful to disable the effects of the TICK interrupt to simplify the environment. This is accomplished by changing the TICK routine to execute an IRET instead of jumping to the dispatcher and not allowing the TICK routine to perform flag set system calls.

When a routine must delay for a specific amount of time, the XIOS usually makes a P_DELAY system call. An example is the delay required after the disk motor is turned on until the disk reaches operational speed. Until the TICK interrupt is implemented, P_DELAY cannot be called and an assembly language time-out loop is needed. To improve performance, replace these time-outs with P_DELAY system calls after the tick routine is implemented and debugged. See the MOTOR_ON: routine in the example XIOS for more details.

8.1 Running Under CP/M-86

To debug Concurrent CP/M under CP/M-86, CP/M-86 must use a console separate from the console used by Concurrent CP/M. Usually a terminal is connected to a serial port and the console input, console output and console status routines in the CP/M-86 BIOS are modified to use the serial port. The serial port thus becomes the CP/M-86 console. Load DDT-86 under CP/M-86 using the remote console and read the CCPM.SYS image into memory using DDT-86. The Concurrent CP/M XIOS must not reinitialize or use the serial port hardware that CP/M-86 is using.

It is somewhat difficult to use DDT-86 to debug an interrupt-driven virtual console handler. Because the DDT-86 debugger operates with interrupts left enabled, unpredictable results can occur.

Values in the CP/M-86 BIOS memory segment table must not overlap
memory represented by the Concurrent CP/M memory partitions
allocated by GENCCPM. CP/M-86, in order to read the Concurrent CP/M
system image under DDT-86, must have in its segment tables the area
of RAM that the Concurrent CP/M system is configured to occupy. See
Figure 8-1.
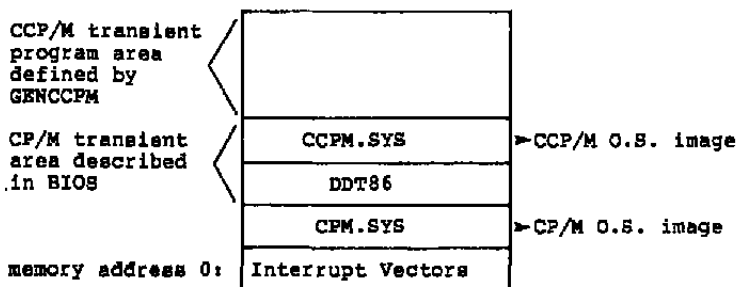
Figure 8-1. Debugging Memory Layout

Any hardware that is shared by both systems is usually not
accessible to CP/M-86 after the Concurrent CP/M initialization code
has executed. Typically, this prevents you from getting out of DDT-
86 and back to CP/M-86, or executing any disk I/O under DDT-86.

The technique for debugging an XIOS with DDT-86 running under CP/M-
86 is outlined in the following steps:

1. Run DDT-86 on the CP/M-86 system.

2. Load the CCPM.SYS file under DDT-86 using the R command and the
    segment address of the Concurrent CP/M system minus 8 (the
    length in paragraphs of the CMD file header). The segment
    address is specified to GENCCPM with the OSSTART option. Set
    up the CS and DS registers with the A-BASE values found in the
    CMD file Header Record. See the Concurrent CP/M Operating
    System Programmer's Reference Guide description of the CMD file
    header.

3. The addresses for the XIOS ENTRY and INIT routines can be found
    in the SYSDAT DATA at offsets 28H for ENTRY and 2CH for INIT.
    These routines will be at offset 0C03H and 0C00H relative to
    the data segment in DS.

4. Begin execution of the CCPM.SYS file at offset 0000H in the
    code segment. Breakpoints can then be set within the XIOS for
    debugging.

In the following figure, DDT-86 is invoked under CP/M-86 and the
file CCPM.SYS is read into memory starting at paragraph 1000H. The
OSSTART command in GENCCPM was specified with a paragraph address of
1008H when the CCPM.SYS file was generated. Using the DDT-86 D(ump)
command the CMD header of the CCPM.SYS file is displayed. As shown,
the A-BASE fields are used for the initial CS and DS segment
register values. The following lines printed by GENCCPM also show
the initial CS and DS values:

        Code starts at 1008
        Data starts at 161A

Two G(o) commands with breakpoints are shown, one at the beginning
of the XIOS INIT routine and the other at the beginning of the ENTRY
routine. These routines can now be stepped through using the the
DDT-86 T(race) command. See the Programmer's Utilities Guide for
more information on DDT-86.

```
A>ddt86
DDT86
-rccpm.sys,1000:0
   START       END
1000:0000 1000:ED7F
-d0
1000:0000 01 12 06 08 10 12 06 00 00 02 B9 08 1A 16 B9 08 .......
                    └──┘                         └──┘
-xcs
CS 0000 1008 ◄────────────┘
DS 0000 161a ◄──────────────────────────────────┘
SS 0051 .
-lds:c00
161A:0C00 JMP       1E2E
161A:0C03 JMP       0C3B

-g,ds:0C00                 ;set a break point at XIOS INIT
*161A:0C00                 ;the INIT routine may now be degugged
   .
   .
   .
-g,ds:0C03                 ;set a break point at XIOS ENTRY
*161A:0C03                 ;the XIOS function being called is
-                          ;AL
-
```

        Figure 8-2. Debugging CCP/M under DDT-86 and CP/M-86

When using SID-86 and symbols to debug the XIOS, extend the CCPM.SYS
file to include unitialized data area not in the file. This ensures
the symbols are not written over while in the debugging session.
Assuming the same CCPM.SYS file as the preceding, use the following
commands to extend the file.

```
SID86
#rccpm.sys,1000:0
  START       END
1000:0000 1000:ED7F
#xcs
CS 0000 1008
DS 0000 161c
SS 0051 .
#sw44
161C:0044 XXXX .                 ;ENDSEG value from SYSDAT DATA
#
#wccpm.sys,1000:0,XXXX:0
#e                               ;release memory
#rccpm.sys,1000:0               ;read in larger file
  START       END
1000:0000 YYYY:ZZZZ
#e*xios                                    ;get XIOS.SYM file
SYMBOLS
#
```

Figure 8-3.   Debugging the XIOS Under SID-86 and CP/M-86

The preceding procedure to extend the file only needs to be
performed once after the CCPM.SYS file is generated by GENCCPM.

End of Section 8

Section 9
Bootstrap Adaptation

This section discusses the example bootstrap procedure for Concurrent CP/M on the IBM Personal Computer. This example is intended to serve as a basis for customization to different hardware environments.

9.1 Components of Track 0 on the IBM PC

Both Concurrent CP/M and CP/M-86 for the IBM Personal Computer reserve track 0 of the 5-1/4 inch floppy disk for the bootstrap routines. The rest of the tracks are reserved for directory and file data. Track 0 is divided into two areas, sector 1 which contains the Boot Sector and sectors 2-8 which contain the Loader. Figure 9-1 shows the layout of track 0 of a Concurrent CP/M boot disk for the IBM Personal Computer.

Sector 1	Boot Sector
Sector 2	Loader
	.
	.
	.
	.
Sector 8	.
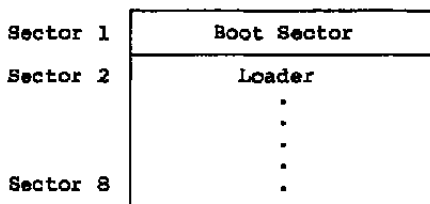
Figure 9-1. Track 0 on the IBM PC

The Boot Sector is brought into memory on reset or power-on by the IBM PC's ROM monitor. The Boot Sector then reads in all of track 0 and transfers control to the Loader.

The Loader is a simple version of Concurrent CP/M that contains sufficient file processing capability to read the CCPM.SYS file, which contains the operating system image, from the boot disk to memory. When the Loader completes its operation, the operating system image receives control and Concurrent CP/M begins execution.

The Loader consists of three modules: the Loader BDOS, the Loader
Program, and the Loader BIOS.   The Loader BDOS is an invariant
module used by the Loader Program to open and read the system image
file from the boot disk.   The Loader Program is a variant module
that opens and reads the CCPM.SYS file, prints the Loader sign-on
message and transfers control to the system image. The Loader BIOS
handles the variant disk I/O functions for the Loader BDOS.   The
term variant indicates that the module is implementation-specific.
The layout of the Loader BDOS, the Loader Program, and the Loader
BIOS is shown in Figure 9-2. The three-entry jump table at 0900H is
used by the Loader BDOS to pass control to the Loader Program and
the Loader BIOS.

Note: The Loader for the IBM PC example begins in sector 2 of
track 0, and continues up to sector 8 along with the rest of the
Loader BDOS, the Loader Program and the Loader BIOS.

              offsets from
              Loader BDOS

                          ┌─────────────────────────┐
                          │   │
                          │   Loader BIOS   │
                          │   │
                          ├─────────────────────────┤
                          │   │
                          │   Loader Program   │
                          │   │
              0909H: │   │
              0906H: │ JMP LOADP   │
              0903H: │ JMP ENTRY   │
              0900H: │ JMP INIT   │
                          │   │
                          │   │
                          │   Loader BDOS   │
                          │   │
              0000H: └─────────────────────────┘
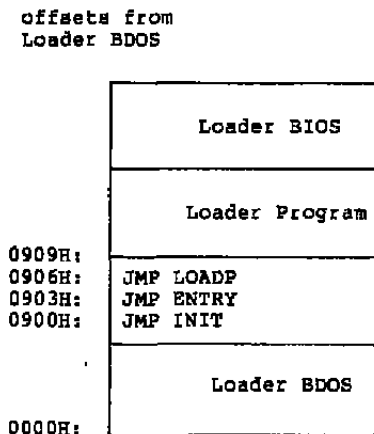
                    Figure 9-2.   Loader Organization
               (Sectors 2 through 8, Track 0 on IBM PC)

9.2   The Bootstrap Process

The sequence of events in the IBM PC after power-on is discussed in
this section.   Except for the functions that are performed by the
IBM ROM monitor, the following process can be generalized to other
8086/8088 machines.

First the ROM monitor reads sector 1, track 0 on drive A: to memory
location 0000:7C00H on power-on or reset.   The ROM then transfers
control to location 0000:7C00H by a JMPF (jump far) instruction.
The Boot Sector program uses the ROM monitor to check for at least
160K of memory contiguous from 0.   The ROM monitor is then used to
read in the remainder of track 0 to memory location 2600:0000H
(152K). Control is transferred to location 2620:0000H, which is the
beginning of the second sector of track 0 and the beginning of the
Loader.   (Each sector is 512 bytes, or 20H paragraphs long.)   The
source code for the Boot Sector program can be found in the file
BOOT.A86 on the Concurrent CP/M distribution disk.

The exact location in memory of the Boot Sector and the Loader
depend on the hardware environment and the system implementor.
However, the Boot Sector must transfer control to the Loader BDOS
with a JMPF (jump far) instruction, with the CS register set to
paragraph address of the Loader BDOS and the IP register set to 0.
Thus the Loader BDOS must be placed on a paragraph boundary. In the
example Loader, the Loader BDOS begins execution with a CS register
set to 2620H and the IP register set to 0000H.

The Loader BDOS sets the DS, SS, and ES registers equal to the CS
register and sets up 64-level stack (128 bytes).   The three Loader
modules, the Loader BDOS, Program and BIOS, execute using an 8080
model (mixed code and data).   It is assumed that the Loader BDOS,
the Loader Program and the Loader BIOS will not require more than 64
levels of stack.   If this is not true then the Loader Program and/or
the Loader BIOS must perform a stack switch when necessary.   The
jump table at 0900H is an invariant part of the Loader, though the
destination offsets of the jump instructions may vary.

After setting up the segment registers and the stack, the Loader
BDOS performs a CALLF (call far) to the JMP INIT instruction at
CS:900H.   The INIT entry is for the Loader BIOS to perform any
hardware initialization needed to read the CCPM.SYS file. Note that
the Loader BDOS does not turn interrupts on or off, so if they are
needed by the Loader, they must be turned on by the Boot Sector or
the Loader BIOS.   The example Loader BIOS executes an STI (Set
Interrupt Enable Flag) instruction in the Loader BIOS INIT routine.

The Loader BIOS returns to the Loader BDOS by executing a RETF
(Return Far) instruction.   The Loader BDOS next initializes
interrupt vector 224 (0E0H) and transfers control to the JMP LOADP
instruction at 0906H, to start execution of the Loader Program.

The Loader Program opens and reads the CCPM.SYS file using the
Concurrent CP/M system calls supported by the Loader BDOS.   The
Loader Program transfers control to Concurrent CP/M through the
"JMPF CCPM" (Jump Far) instruction at the end the Loader Program,
thus completing the loader sequence. The following sections discuss
the organization of the CCPM.SYS file and the memory image of
Concurrent CP/M.

9.3 The Loader BDOS and Loader BIOS Function Sets

The Loader BDOS has a minimum set of functions required to open the
system image file and transfer it to memory. These functions are
invoked as under Concurrent CP/M by executing a INT 224 (00E0H) and
are documented in the Concurrent CP/M Programmer's Reference Guide.
The functions implemented by the Loader BDOS are in the following
list. Any other function, if called, will return a 0FFFFh error
code in registers AX and BX.

Func#	CL	Function Name
14	0Eh	Select Disk
15	0Fh	Open File
20	14h	Read Sequential
26	1Ah	Set DMA Offset
32	20h	Set/Get User Number
44	2Ch	Set Multisector Count
51	33h	Set DMA Segment

Blocking/Deblocking has been implemented in the Loader BDOS, as well
as multisector disk I/O. This simplifies writing and debugging the
loader BIOS and improves the system load time. File LBDOS.H86
includes the Loader BDOS.

The Loader BIOS must implement the minimum set of functions required
by the Loader BDOS to read a file.

Func#	AL	Function Name
9	09H	IO_SELDSK (select disk)
10	0AH	IO_READ (read physical sectors)

To invoke IO_SELDSK or IO_READ in the Loader BIOS, the Loader BDOS
performs a CALLF (Call Far) instruction to the jump instruction at
ENTRY (0903H).

The Loader BIOS functions are implemented in the same way as the
corresponding XIOS functions. Therefore the code used for the
Loader BIOS may, with a few exceptions, be a subset of the system
XIOS code. For example, the Loader BIOS does not use the
DEV_WAITFLAG or DEV_POLL Concurrent CP/M system functions. Certain
fields in the Disk Parameter Headers and Disk Parameter Blocks can
be initialized to 0, as in Figure 9-3:

Disk Parameter Header

00H	XLT	0000	00	00	0000
08H	DPB	0000	0000		DIRBCB
10H	DATBCB	0000			

Disk Parameter Block

00H	SPT		BSH	BLM	EXM	DSM	DRM...
08H	..DRM	00	00	0000		OFF	PSH
10H	PHM						

Figure 9-3.   Disk Parameter Field Initialization

The Loader Program and Loader BIOS may be written as separate modules, or combined in a single module as in the example Loader. The size of these two modules can vary as dictated by the hardware environment and the preference of the system implementor.   The LOAD.A86 file contains the Loader Program and the Loader BIOS. LOAD.A86 appears on the Concurrent CP/M release disk, and may be assembled and listed for reference purposes.

The Loader Program and the Loader BIOS are in a contiguous section of the Loader to reduce the size of the Loader image. Grouping the variant code portions of the Loader into a single module, allows the implementation of nonfile-related functions in the most size-efficient manner. The example Loader BIOS implements the IO_CONOUT function in addition to IO_SELDSK and IO_READ. This Loader BIOS can be expanded to support keyboard input to allow the Loader Program to prompt for user options at boot time. However, the only Loader BIOS functions invoked by the Loader BDOS are IO_SELDSK and IO_READ, any other Loader BIOS functions must be invoked directly by the Loader Program.

9.4 Track 0 Construction

Track 0 for the example IBM PC bootstrap is constructed using the following procedure: The Boot Sector is 0200H (512) bytes long and is assembled with the command:

    A>ASM86 BOOT

This results in the file BOOT.H86, which becomes a binary CMD file with the command:

    A>GENCMD BOOT 8080

The LOAD.A86 file, containing the the Loader Program and the Loader
BIOS, is assembled using the command:

    A>ASM86 LOAD

The Loader BDOS starts a 0000H and ends at 0900H. The LOAD module
starts at 0900H and ends at 0E00H. This equals the size of the 7
sectors remaining after the Boot Sector. The IBM PC disk format has
eight 0200H-byte (512-byte) sectors, or 1000H (4K) bytes per track.
Subtracting 0200H, the length of the Boot Sector, we get 0E00H. The
LOADER.H86 file, containing the Loader BIOS, Loader Program and
Loader BIOS, is constructed using the command:

    A>PIP LOADER.H86=LBDOS.H86,LOAD.H86

Next a binary CMD file is created from LOADER.H86 with GENCMD:

    A>GENCMD LOADER 8080

This results in the file LOADER.CMD with a header record defining
the 8080 Model. Note this CMD file is not directly executable under
any CP/M operating system, but can be debugged as outlined below.
Next the BOOT.CMD and LOADER.CMD files are combined into a track
image. Use DDT-86 or SID-86 to do this:

```
A>DDT86                      ; or SID86
-rboot.cmd
  START       END            ; aaaa is paragraph where DDT86
aaaa:0000 aaaa:027F          ; places BOOT.CMD
-wtrack0,80,107f             ; create the 4K file, TRACK0, without
                             ; a CMD header
-rtrack0                     ; read the 4K TRACK0 file into memory
  START       END
-bbbb:0000 bbbb:0FFF         ; TRACK0 starts at paragraph bbbb
-rloader.cmd                 ; read LOADER.CMD to another area of
  START       END            ; memory
-zzzz:0000 zzzz:0E7F         ; LOADER.CMD starts at paragraph zzzz
-mzzzz:80,0E7F,bbbb:0200     ; move the Loader to where sector 2
                             ; starts in the track image
-wtrack0,bbbb:0,0FFF         ; write the track image to the file
                             ; TRACK0
```

The final step is to place the contents of TRACK0 onto track 0. The
TCOPY example program accomplishes this with the following command:

    A>TCOPY TRACK0

Scratch diskettes should be used for testing the Boot Sector and
Loader. TCOPY is included as the source file TCOPY.A86, and needs
to be modified to run in hardware environments other than the IBM
PC.   TCOPY only runs under CP/M-86 and cannot be used under
Concurrent CP/M.

The Loader can be debugged separately from the Boot Sector under
DDT-86 or SID-86, using the following commands:

```
A>DDT86                      ; or SID86
-rloader.cmd
  START      END             ; aaaa is paragraph where DDT86
aaaa:0000 aaaa:0E7F          ; places the Loader
-haaaa,8                     ; Add 8 paragraphs to skip over CMD
yyyy,zzzz                    ; header, aaaa + 8 = yyyy
-xcs
CS 0000 yyyy                 ; set CS for debugging
-1900                        ; IP is set to 0 by DDT86 or SID86
  ...
  ...
  ...
```

The 1900 command lists the jumps to INIT, ENTRY and LOADP to verify
the Loader Program and the Loader BIOS are at the correct offsets.
Breakpoints can now be set in the Loader Program and Loader BIOS.
The Boot Sector can be debugged in a similar manner, but sectors 2
through 8 need to contain the Loader image if the JMPF LOADER
instruction in the Boot Sector is to be executed.

9.5 Other Bootstrap Methods

The preceding three sections outline the operation and steps for
constructing a bootstrap loader for Concurrent CP/M on the IBM PC.
Many departures from this scheme are possible and they depend on the
hardware environment and the goals of the implementor. The Boot
Sector can be eliminated if the system ROM (or PROM) can read in the
entire Loader at reset.   The Loader can be eliminated if the
CCPM.SYS file is placed on system tracks and the ROM can read in
these system tracks at reset. However, this scheme usually requires
too many system tracks to be practical. Alternatively, the Loader
can be placed into a PROM and copied to RAM at reset, eliminating
the need for any system tracks. If the Boot Sector and the Loader
are eliminated, any initialization normally performed by the two
modules must be performed in the XIOS initialization routine.

9.6   Organization of CCPM.SYS

The CCPM.SYS file, generated by GENCCPM and read by the Loader,
consists of the seven *.CON files and any included *.RSP files. The
CCPM.SYS file is prefixed by a 128-byte CMD Header Record, which
contains the following two Group Descriptors:

G-Form	G-Length	A-Base	G-Min	G-Max
01h	xxxx	1008h	xxxx	xxxx
02h	xxxx	(varies)	xxxx	xxxx

Figure 9-4.   Group Descriptors - CCPM.SYS Header Record

The first Group Descriptor represents the O.S. Code Group of the
CCPM.SYS file and the second represents the Data. The preceding
Code Group Descriptor has an A-Base load address at paragraph 1008H,
or "paragraph:byte" address of 01008:0000H. The A-Base value in the
Data Group Descriptor varies according to the modules included in
this group by GENCCPM. The load address value shown above is only
an example. The CCPM.SYS file can be loaded and executed at any
address where there is sufficient memory space. The entire CCPM.SYS
file appears on disk as shown in Figure 9-5.

Image in Memory   Image in CCPM.SYS

(High Memory)

ENDSEG

Disk Buffers

(End of File)

| RSPs
(including TMP, CLOCK) |
| --- |

RSPSEG

O.S. Table Space

| CCPM.SYS
DATA
GROUP |
| --- |

System
Data
Area

XIOS Code and Data

──OCOOH──
(XIOS)

O.S. Data

──XIOS──
(CS:,DS:)

O.S. Code

| CCPM.SYS
CODE
GROUP |
| --- |

OSSEG

| CCPM.SYS
HEADER |
| --- |

Low Memory

(Start of File)
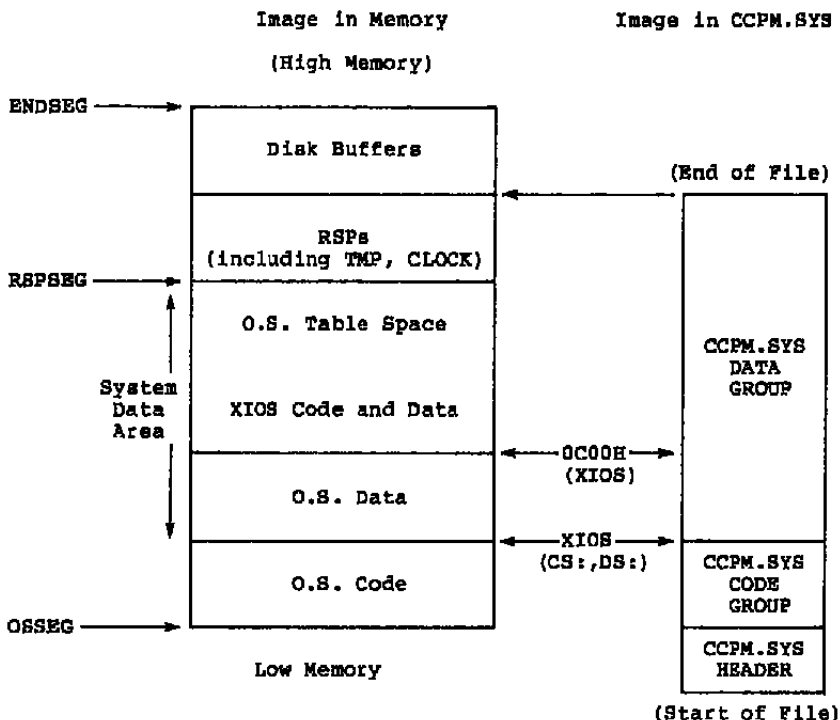
Figure 9-5.   CCPM System Image and the CCPM.SYS File

The CCPM.SYS file is read into memory by the Loader beginning at the
address given by Code Group A-Base (in the example shown above,
paragraph address 1008H), and control is passed to the Supervisor
INIT function when the Loader Program executes a JMPF instruction
(Jump Far) to 1008:0000H. The Supervisor INIT must be entered with
CS set to the value found in the A-BASE field of the code Group
Descriptor, the IP register equal to 0 and the DS register equal to
A-BASE value found in the data Group Descriptor.

End of Section 9

Section 10
OEM Utilities

A commercially viable Concurrent CP/M system requires OEM-supported
capabilities. These capabilities include methods for formatting
disk and image backups of disks. Typically, an OEM supplies the
following utilities:

- Disk Formatting Utility (FORMAT.CMD)
- Disk Copy Utility (DCOPY.CMD)

These utilities are usually hardware-specific and either make direct
XIOS calls or go directly to the hardware.

10.1 Bypassing the BDOS

When special OEM utilities bypass the BDOS by making direct XIOS
calls or going directly to the hardware, several programming
precautions are necessary to prevent conflicts due to the Concurrent
CP/M multitasking environment. The following steps must be taken to
prevent other processes from accessing the disk system:

1. Warn the user. This program bypasses the operating system. No
   other programs should be running while this program is being
   used.

2. Check for Version 2 or 3.1 of Concurrent CP/M through the
   S_OSVER function. The following steps are specific to these
   versions of Concurrent CP/M. They do not work in previous
   Digital Research operating systems, nor are they guaranteed to
   work in future Digital Research operating systems.

3. Set the process priority to 150 or better through the
   P_PRIORITY function. If another program is running on a
   background console, it cannot obtain the CPU resource while
   this program needs it.

4. Set the P_KEEP flag in the Process Descriptor to prevent
   termination of the operation without proper cleanup.

5. Make sure the program is running in the foreground and that the
   console is in DYNAMIC mode. Then lock the console into the
   foreground by setting the NOSWITCH flag in the CCB. This
   prevents the user from initiating a program on another virtual
   console while this program is running in the background.
   Because the file system is locked, a program cannot load from
   disk.

6. Make sure there are no open files in the system. This also
   detects background virtual consoles in BUFFERED mode.

7. Lock the BDOS by reading the MXdisk queue message.

8. You can now safely perform the FORMAT and DCOPY operations on the disk system, independent of the BDOS.

9. Once the operations are complete, allow the disk system to be reset by setting the login sequence number in each affected DPH to 0. When the disk system is reset, these drives are reset even if they are permanent. The login sequence field is 06h bytes from the beginning of the DPH.

10. Release the BDOS by writing the MXdisk queue message.

11. Reset the Disk System with the DRV_ALLRESET function.

12. Unlock the console system allowing console switching by unsetting the NOSWITCH bit of the CCB_FLAG field in the CCB.

13. Reset the P_KEEP flag in the Process Descriptor.

14. Terminate.

Listing 10-1 illustrates these steps and shows how to make direct XIOS calls to access the disk system. The routines corresponding to the steps are labeled for cross-reference purposes.

```
PAGEWIDTH          80
;
;***************************************************
;*
;*        PHYSICAL.A86
;*
;*        Sample Program Illustrating Direct Calls to
;*        the Disk Routines in the XIOS.
;*
;*        This program will lock the console and disk
;*        systems, read a physical sector into memory
;*        and gracefully terminate.
;*
;***************************************************

true               equ       0ffffh
false              equ       0

cr                 equ       0dh
lf                 equ       0ah

ccpmint            equ       224
ccpmver2           equ       01420H

          ; XIOS functions

io_seldsk          equ       09h
io_read            equ       0ah
io_write           equ       0bh

          ; SYSDAT Offsets

sy_xentry          equ       028h
sy_nvcns           equ       047h
sy_ccb             ·equ      054h
sy_openfile        equ       088h

          ; Process Descriptor
p_flag             equ       word ptr 06h
p_uda              equ       word ptr 010h
pf_keep            equ       00002h

          ; Console Control Block
ccb_size           equ       02ch
ccb_state          equ       word ptr 0eh
cf_buffered        equ       00001h
cf_background      equ       00002h
cf_noswitch        equ       00008h
```

Listing 10-1. Disk Utility Programming Example

```
        ; Disk Parameter Header

dph_lseq          equ      byte ptr 06h

        ; drvvec bits

drivea            equ      00001h
driveb            equ      00002h
drivec            equ      00004h

;****************************************************
;*
;*      CODE SEGMENT
;*
;****************************************************

        CSEG
        ORG     0

        ; Switch Stacks to make sure we have enough.
        ; This is done with interrupts off.
        ; Old 8086's and 8088's will allow an
        ; interrupt between SS and SP setting.

        pushf | pop bx
        cli
        mov ax,ds | mov ss,ax
        mov sp,offset tos
        push bx | popf

        ; Step 1. - Warn the user.

        mov dx,warning | call c_writebuf

        ; Step 2. - Check for Concurrent CP/M V3.1

        call s_osver
        and ax,0fff0h
        cmp ax,ccpmver2 | je good_version
            jmp bad_version
good_version:

        ; Step 3 - Set priority to 150

        mov dl,150
        call p_priority          ;priority = 150

        call get_osvalues        ;get OS values
```

Listing 10-1. (continued)

```
        ; Step 4 - Set the P_KEEP flag in PD

        call no_terminate        ;set p_keep flag

        ; Step 5 - Lock the console

        call lock_con            ;lock consoles

        ; Step 6 and 7 - Lock the BDOS,
        ;       make sure there are no open files

        call lock_disk           ;lock bdos

        ; Step 8 - Perform the Operation

        call operation           ;do operation

        jmp terminate            ;terminate

operation:
;----------
; Do our disk operations.  If we make changes to a
; disk, make sure to set the appropriate bit in the
; drvvec variable to force the BDOS to reinitialize
; the drive.  In this example are only going to
; read a physical sector from disk.

        ; Lets read Track 2 Sector 2 of drive B
        ; with DMA set to sectorbuf
        ; Setup for Direct IO_READ call with
        ; IOPB on Stack.

        mov ax,ds                ;save for DMA seg
        push es ! push ds
        mov es,udaseg
        mov ds,sysdat
        mov ch,1                 ;mscnt = 1
        mov cl,1 ! push cx       ;drive = B
        mov cx,2 ! push cx       ;track = 2
        mov cx,2 ! push cx       ;sector = 2
        push ax                  ;DMA Seg = Our DS
        mov cx,offset sectorbuf
        push cx                  ;DMA Ofst
        mov ax.io_read
                ; do the read
        callf dword ptr .sy_xentry
        add sp,10
        pop ds ! pop es
        cmp al,0 ! je success
            mov dx,offset physerr
            call c_writebuf
```

Listing 10-1. (continued)

```
success:
                ; force a keystroke to allow testing
                ; of locking mechanisms
        jmp c_read

get_osvalues:
;-------------
; get system addresses for later use

                ; Get System Data Area Segment
        push es
        call s_sysdat
        mov sysdat,es

                ; Get Process Descriptor Address
        call p_pdadr
        mov pdaddr,bx

                ; Get User Data Area Segment for
                ; XIOS calls
        mov ax,es:p_uda[bx]
        mov udaseg,ax
        pop es
        ret

no_terminate:
;------------
; Set the pf_keep flag.  We cannot be terminated.

        mov bx,pdaddr
        push ds | mov ds,sysdat
        or p_flag[bx],pf_keep
        pop ds
        ret

lock_disk:
;---------
; Lock the BDOS.  No BDOS calls will be allowed in
; the system until we unlock it.

                ;get currently logged in drives
                ;for later reset
        call drv_loginvec
        mov drvvec,ax
                ;read mxdisk queue message
        mov dx,offset mxdiskqpb | call q_open
        mov dx,offset mxdiskqpb | call q_read
                ;turn on bdoslock flag for
                ;terminate
        mov bdoslock,true
```

Listing 10-1. (continued)

10-6

```
                      ;verify no open files.  This will
                      ;also check background consoles in
                      ;buffered mode since they have open
                      ;files when active.
              push ds 1 mov ds,sysdat
              cmp word ptr .sy_openfile,0
              pop ds
              je lckb
                      ;Error, open files
                  jmp openf
lckb:     ret

bdos_unlock:
;-----------
; unlock the BDOS.  Reset all logged in drives to
; make sure BDOS reinitializes them internally.

                      ;reset all loggedin drives as well
                      ;as drives we have played with.
              xor cx,cx
              mov ax,drvvec
resetd: cmp cx,16 1 je rdone
              test ax,1 1 jz nextdrv
                          ; we have a logged in drive,
                          ; get DPH address from XIOS
                  push cx 1 push ax
                  push es 1 push ds
                  mov es,udaseg
                  mov ds,sysdat
                  mov ax,io_seldsk
                  mov dx,0
          '       callf dword ptr .sy_xentry
                          ; if legal drive, set
                          ; login sequence # to 0.
xret:             cmp bx,0 1 je nodisk
                  mov dph_lseq[bx],0
nodisk:           pop ds 1 pop es
                  pop ax 1 pop cx
                          ;try another drive
nextdrv:      inc cx
              shr ax,1
              jmps resetd
                  ; all drives can be reset,
                  ; write mxdisk queue message
                  ; reset all drives
rdone:  mov dx,offset mxdiskqpb
        call q_write
        jmp drv_resetall
```

Listing 10-1.   (continued)

```
lock_con:
;---------
; Lock the console system

        call getccbadr
        mov bx,ccbadr
        push ds | mov ds,sysdat
        pushf | cli
                ; make sure our console is
                ; foreground, dynamic
        cmp ccb_state[bx],0 | je foreg
            popf | pop ds
            jmp in_back

foreg:
                ; set console to NOSWITCH
        or ccb_state[bx],cf_noswitch
        popf | pop ds
                ; turn on conlock flag for
                ; terminate
        mov conlock,true
        ret

con_unlock:
;-----------
; Set console to switchable.

        mov bx,ccbadr
        push ds | mov ds,sysdat
        and ccb_state[bx],not cf_noswitch
        pop ds
        ret

getccbadr:
;---------
; Calculate the CCB address for this console.

        call c_getnum
        xor ah,ah
        mov cx,ccb_size | mul cx
        push ds | mov ds,sysdat
        add ax,.sy_ccb
        pop ds
        mov ccbadr,ax
        ret

bad_version:
;-----------
        mov dx,offset wrong_version
        jmps errout
```

<div align="center">Listing 10-1.   (continued)</div>

```
in_back:
;--------
        mov dx,offset in_background
        jmps errout
openf:
;-----
        mov dx,offset openfiles
errout:
        call c_writebuf
terminate:
;---------

        ; Step 9,10,11  Clean up the file system

        cmp bdoslock,false ! je t01
            call bdos_unlock

        ; Step 12 - Unlock the console system

t01:    cmp conlock,false ! je t02
            call con_unlock

        ; Step 13 - Unset the P_KEEP flag in PD

t02:    mov bx,pdaddr
        push ds ! mov ds,sysdat
        and p_flag[bx],not pf_keep
        pop ds

        ; Step 14 - Terminate

        jmp p_termcpm

;--------------
; OS functions
;--------------

c_getnum:       mov cl,153 ! jmps ccpm
c_read:         mov cl,1 ! jmps ccpm
c_writebuf:     mov cl,9 ! jmps ccpm
drv_loginvec:   mov cl,24 ! jmps ccpm
drv_resetall:   mov cl,13 ! jmps ccpm
p_pdadr:        mov cl,156 ! jmps ccpm
p_priority:     mov cl,145 ! jmps ccpm
p_termcpm:      mov cl,0 ! jmps ccpm
q_open:         mov cl,135 ! jmps ccpm
q_read:         mov cl,137 ! jmps ccpm
q_write:        mov cl,139 ! jmps ccpm
s_osver:        mov cl,163 ! jmps ccpm
s_sysdat:       mov cl,154 ! jmps ccpm
ccpm:           int ccpmint
                ret
```

Listing 10-1. (continued)

```
;******************************************************
;*
;*        DATA SEGMENT
;*
;******************************************************

          DSEG
          ORG     0100H

sysdat          dw      0
pdaddr          dw      0
udaseg          dw      0
ccbadr          dw      0
drvvec          dw      0
bdoslock        db      false
conlock         db      false

mxdiskqpb       dw      0,0,0,0
                db      'MXdisk  '

          ;  ERROR MESSAGES

warning         db      'PHYSICAL: This program '
                db      'bypasses the operating '
                db      'system.',cr,lf
                db      'Make sure no other '
                db      'programs are running.'
                db      cr,lf,'$'

in_background   db      'PHYSICAL: must be run '
                db      'in the foreground, in'
                db      ' DYNAMIC mode.',cr,lf,'$'

wrong_version   db      'PHYSICAL: runs only on '
                db      'Concurrent CP/M Version 2'
                db      cr,lf,'$'

open_files      db      'PHYSICAL: cannot run'
                db      'while there are open files.'
                db      cr,lf
                db      'If any virtual consoles are'
                db      ' in BUFFERED mode,',cr,lf
                db      'Use the VCMODE D command to'
                db      ' set a virtual console to '
                db      'DYNAMIC mode.',cr,lf,'$'

physerr         db      'Physical Error on Read.'
                db      cr,lf,'$'

sectorbuf       rb      1024
```

Listing 10-1.   (continued)

```
        ; Lots of Stack.  Bottom prefilled with 0cch
        ; (INT 3 instruction) to see if we are
        ; overrunning the stack.  Also if we
        ; accidently execute it under DDT86,
        ; a breakpoint occurs.

        DW      0CCCCH,0CCCCH,0CCCCH
        DW      0CCCCH,0CCCCH,0CCCCH
        DW      0CCCCH,0CCCCH,0CCCCH
        DW      0CCCCH,0CCCCH,0CCCCH
        DW      0CCCCH,0CCCCH,0CCCCH
        DW      0CCCCH,0CCCCH,0CCCCH
        DW      0CCCCH,0CCCCH,0CCCCH
        DW      0CCCCH,0CCCCH,0CCCCH

        RW      0100H
tos     DW      0CCCCH          ; DW at end of DATA SEG
                                ; to make sure HEX is
                                ; generated.

        END     ; End of PHYSICAL.A86
```

;--

Listing 10-1. (continued)

10.2 Directory Initialization in the FORMAT Utility

The FORMAT utility initializes fresh disk media for use with
Concurrent CP/M. It is written by the OEM and packaged with
Concurrent CP/M as a system utility. The physical formatting of a
disk is hardware-dependent and therefore is not discussed here.
This section discusses initialization of the directory area of a new
disk.

The FORMAT program can initialize the directory with or without time
and date stamping enabled. This can be a user option in the FORMAT
program. If time and date stamps are not initialized, the user can
independently enable this feature through the INITDIR and SET
utilities.

It is highly recommended that the OEM supports the advanced features
of Concurrent CP/M including time and date stamping in the FORMAT
program. This allows the user to use these features in their
default disk format. Otherwise, the user must first learn that date
stamps are possible and then must use the INITDIR and SET utilities
to allow the use of this feature. If the disk directory is too
close to being full, the INITDIR program will not allow the
restructuring of the directory that is necessary to include SFCB's.

The cost of enabling the time and date stamp feature on a given disk
is 25% of its total directory space. This space is used to store
the time and date information in special directory entries called
SFCBs. For time and date stamping, every fourth directory entry
must be an SFCB. Each SFCB is logically an extension of the
previous three directory entries. This method of storing date-stamp
information allows efficient update of date stamps since all of the
directory information for a given file resides within a single 128-
byte logical disk record.

A disk under Concurrent CP/M is divided into three areas, the
reserved tracks, the directory area and the data area. The size of
the directory and reserved areas is determined by the Disk Parameter
Block, described in Section 5.5. The data area starts on the first
disk allocation block boundary following the directory area.

```
+-------------------------+
|    Reserved Tracks      |
+-------------------------+
|    Directory Area       |
+-------------------------+|
|                         |
|       Data Area         |
|                         |
+-------------------------+
```
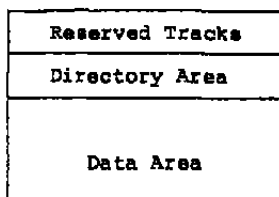
Figure 10-1. Concurrent CP/M Disk Layout

The reserved area and the data area do not need to be initialized to
any particular value before use as a Concurrent CP/M disk. The
directory area, on the other hand, must be initialized to indicate
that no files are on the disk. Also, as discussed below, the FORMAT
program can reserve space for time and date information and
initialize the disk to enable this feature.

The directory area is divided into 32-byte structures called
Directory Entries. The first byte of a Directory Entry determines
the type and usage of that entry. For the purposes of directory
initialization, there are three types of Directory Entries that are
of concern: the unused Directory Entry, the SFCB Directory Entry
and the Directory Label.

A disk directory initialized without time and date stamps has only
the unused type of Directory Entry. An unused Directory Entry is
indicated by a 0E5H in its first byte. The remaining 31 bytes in a
Directory Entry are undefined and can be any value.

```
             OH     1H                                  20H
  entry 0 |  0E5H |          undefined                     |
        1 |  0E5H |            .                           |
        2 |  0E5H |            .                           |
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

                                .
                                .
                                .
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        n |  0E5H |          undefined                     |
```
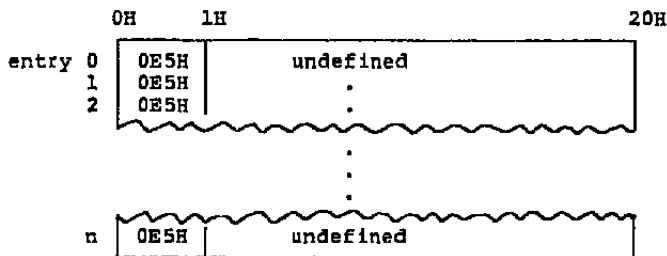
Figure 10-2.   Directory Initialization Without Time Stamps

A disk directory initialized to enable time and date stamps must
have SFCB's as every fourth Directory Entry. An SFCB has a Q21H in
the first byte and all other bytes must be 0H. Also a directory
label must be included in the directory. This is usually the first
Directory Entry on the disk.   The directory label must be
initialized as shown in Figure 10-3.

```
 OH    1H              OCH   0DH   0EH   0FH   10H
| 20H |  NAME   }  { | DATA | 00H | 00H | 00H |

 10H   11H   12H   13H   14H   15H   16H   17H   18H
| 20H | 20H | 20H | 20H | 20H | 20H | 20H | 20H |

 18H   19H   1AH   1BH   1CH   1DH   1EH   1FH   20H
| 00H | 00H | 00H | 00H | 00H | 00H | 00H | 00H |
```
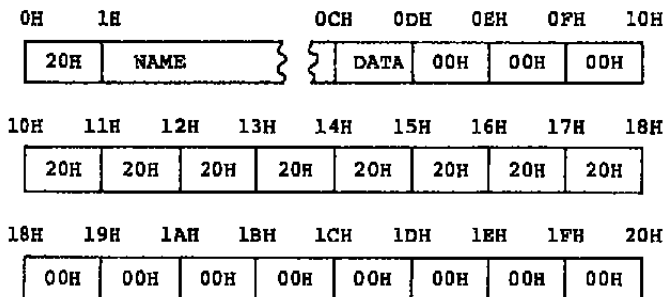
Figure 10-3.   Directory Label Initialization

Table 10-1. Directory Label Data Fields

Field	Explanation
NAME	An 11 byte field containing an ASCII name for the drive. Unused bytes should be initialized to blanks (20H).
DATA	A bit field that tells the BDOS general characteristics of files on the disk. The DATA field can assume the following values: • 060H enables date of last modification and date of last access to be updated when appropriate. • 030H enables date of last modification and date of creation to be updated when appropriate.

The FORMAT program should ask the user for the name of the disk and whether to use the date of last access or the date of creation for files on this disk. The date of last modification should always be used. If the DATA field is 0H or if the Directory Label does not exist, the time and date feature is not enabled. The DATA Field must be 0H if SFCB's are not initialized in the directory.

```
           0H       1H                                      20H

entry  0  | 020H   | NAME,DATA    (Directory Label) |
       1  | 0E5H   | undefined    (Unused)           |
       2  | 0E5H   | undefined    (Unused)           |
       3  | 021H   | NULLS        (SFCB)             |
       4  | 0E5H   | undefined    (Unused)           |
       5  | 0E5H   | undefined    (Unused)           |
       6  | 0E5H   | undefined    (Unused)           |
       7  | 021H   | NULLS        (SFCB)             |

                               .
                               .
                               .

          | 0E5H   | undefined    (Unused)           |
          | 0E5H   | undefined    (Unused)           |
          | 0E5H   | undefined    (Unused)           |
       n  | 021H   | NULLS        (SFCB)             |
```

Figure 10-4.   Directory Initialization With Time Stamps

End of Section 10

Section 11
End-user Documentation

OEMs must be aware that the documentation supplied by Digital Research for the generic release of Concurrent CP/M describes only the example XIOS implementation. If the OEM decides to change, enhance, or eliminate a function which impacts the Concurrent CP/M operator interface, he must also issue documentation describing the new implementation. This is best done by purchasing reprint rights to the Concurrent CP/M system publications, rewriting them to reflect the changes, and distributing them along with the OEM-modified system.

One area that is highly susceptible to modification by the OEM is the Status Line XIOS function. Depending upon the implementation, it might be desirable to display different, more, or even no status parameters. The documentation supplied with Concurrent CP/M, however, assumes that the Status Line function is implemented exactly like the example XIOS presented herein.

Another area which the OEM might want to change is the default login disk. At system boot time, the default system disk as specified in the system GENCCPM session is automatically logged-in and displayed in the first system prompt. However, a startup command file, STARTUP.N, where N is the Virtual Console number, can be implemented for each Virtual Console. This file can switch the default logged-in disk drive to any drive desired. However, the Concurrent CP/M Operating System User's Guide assumes that the prompt will show the system disk. For more information on startup files, see the Concurrent CP/M Operating System User's Guide and the Concurrent CP/M Operating System Programmer's Reference Guide.

The Concurrent CP/M system prompt is similar to the CP/M 3 prompt in that the User Number is not displayed for User 0. If the user changes to a higher User Number, then the User Number is displayed as the first character of the prompt, for example 5A>. If the OEM wants to change this, or any other function of the user interface, such as implementing Programmable Function Keys, he can rewrite the TMP module source code included with the system. However, documenting these changes is entirely the OEM's responsibility.

End of Section 11

Appendix A
Removable Media

All disk drives are classified under Concurrent CP/M as having either permanent or removable media. Removable-media drives support media changes; permanent drives do not. Setting the high-order bit of the CKS field of the drive's DPB marks the drive as a permanent-media drive. See Section 5.5, "Disk Parameter Block."

The BDOS file system makes two important distinctions between permanent and removable-media drives. If a drive is permanent, the BDOS always accepts the contents of physical record buffers as valid. It also accepts the results of hash table searches on the drive.

BDOS handling of removable-media drives is more complex. Because the disk media can be changed at any time, the BDOS discards directory buffers before performing most system calls involving directory searches. By rereading the disk directory, the BDOS can detect media changes. When the BDOS reads a directory record, it computes a checksum for the record and compares it to the current value in the drive's checksum vector. If the values do not match, the BDOS assumes the media has been changed, aborts the system call routine, and returns an error code to the calling process. Similarly, the BDOS must verify an unsuccessful hash table search for a removable-media drive by accessing the directory. The point to note is that the BDOS can only detect a media change by reading the directory.

Because of the frequent necessity of directory access on removable-media drives, there is a considerable performance overhead on these drives compared to permanent drives. Another disadvantage is that, since the BDOS can detect media removal only by a directory access, inadvertantly changing media during a disk write operation results in writing erroneous data onto the disk.

If, however, the disk drive and controller hardware can generate an interrupt when the drive door is opened, another option for preventing media change errors becomes available. By using the following procedure, the performance penalty for removable-media drives is practically eliminated.

1. Mark the drive as permanent by setting the value of the CKS field in the drive's DPB to 8000H plus the total number of directory entries divided by 4. For example, you would set the CKS for a disk with 96 directory entries to 8018H.

2. Write a Door Open Interrupt routine that sets the DOOR field in the XIOS Header and the DPH Media Flag for any drive signalling an open door condition.

A-1

The BDOS checks the XIOS Header DOOR flag on entry to all disk-
related XIOS function calls. If the DOOR flag is not set, the BDOS
assumes that the removable media has not been changed. If the DOOR
flag is set (0FFH), the BDOS checks the Media Flag in the DPH of
each currently logged-in drive. It then reads the entire directory
of the drive to determine whether the media has been changed before
performing any operations on the drive. The BDOS also temporarily
reclassifies the drive as a removable-media drive, and discards all
directory buffers to force all subsequent directory-related
operations to access the drive.

In summary, using the DOOR and Media Flag facilities with removable-
media drives offers two important benefits. First, performance of
removable-media drives is enhanced. Second, the integrity of the
disk system is greatly improved because changing media can at no
time result in a write error.

                         End of Appendix A

Appendix B
Graphics Implementation

Concurrent CP/M can support graphics on any virtual console assigned
to a physical console that has graphics capabilities. Support is
provided in the operating system for GSX, that has its own separate
I/O system, GIOS. The GIOS does its own hardware initialization to
put a physical console in graphics mode. A graphics process that is
in graphics mode can not run on a background console, because this
would cause the foreground console to change to graphics mode.
Also, whenever the foreground console is initialized for graphics,
you cannot switch the screen to another virtual console. The
following points need to be kept in mind when writing an XIOS for a
system that will support graphics.

- IO_SCREEN (Function 30) will be called by the GIOS when it
  wants to change a virtual console to graphics or alphanumeric
  mode. If the virtual console is in the background and graphics
  is requested, IO_SCREEN must flagwait the process. If the
  virtual console is in the foreground, change the screen_mode
  and allow the process to continue. You must reserve at least
  one flag for each virtual console for this purpose. See
  Section 6.1 "Screen I/O Functions" for more information on
  IO_SCREEN.

- IO_SWITCH (Function 7) must flagset any process that was
  flagwaited by IO_SCREEN when its virtual console is switched to
  the foreground. When a foreground console is in graphics mode,
  IO_SWITCH will not be called, because PIN calls Function 30
  (get), ignoring the switch key if the screen is in graphics
  mode. Thus while a graphics process is running in graphics
  mode in the foreground, it is not possible to switch screens.
  For more information on IO_SWITCH see Section 4.2 "Console I/O
  Functions".

- IO_STATLINE (Function 8) must not display the status line on a
  console that is in graphics mode. This can be done by checking
  the same variable in the screen structure that Function 30
  returns as the screen mode. For more information on
  IO_STATLINE see Section 4.2 "Console I/O Functions".

End of Appendix B

Index

XIOS
  8080 Model, 1-4
  debugging, 8-1
  reentrant code, 1-13
  relationship to CCPM.SYS
    file, 1-4
  spurious interrupt handling,
    3-9
XIOS.CON, 2-2
XLT, 5-21
XPCNS, 1-20, 4-2

NOTES