CP/M-86 Plus

Operating System

Installation Guide

Digital Research
P.O. Box 579
160 Central Avenue
Pacific Grove, CA 93950
TWX 910 360 5001

Foreword

CP/M-86 Plus is a single-user, multitasking operating system. It is designed for use with any disk-based microcomputer using an Intel 8086 or 8088 microprocessor. This CP/M-86 Plus Operating System Installation Guide, Release 3.0, (hereinafter cited as the Installation Guide) is intended to assist system implementers and OEMs who are porting CP/M-86 Plus to a new 8086/8088-based computer.

The central task in porting or customizing CP/M-86 Plus is the creation of a Basic Input/Output System (BIOS) for the target machine. Sections 1-10 cover the development of the BIOS.

If you are unfamilar with customizing Digital Research operating systems, Appendix A breaks down BIOS development into a series of steps or base levels. You can use Appendix A as a starting point for planning your own series of steps to create an operational BIOS.

Part of porting CP/M-86 Plus is the generation of a bootstrap loader to initially bring CP/M-86 Plus into memory at power-on or hardware reset. Porting additionally involves the creation of any needed hardware-dependent utilities, such as a disk formatter. Bootstrap and loading can be accomplished before the BIOS since they entail writing a simplified loader BIOS, which can then be expanded into the full BIOS. Another possibility is to leave the loader to last and strip down the full BIOS into the loader BIOS. Sections 11 and 12 discuss bootstrap operations and hardware-dependent utilities respectively.

The appendixes cover the optional tasks of placing CP/M-86 Plus in ROM, customizing the Console Command Processor (CCP), and changing system and utility messages to other languages.

You need the following software and hardware for porting CP/M-86 Plus:

- CP/M-86 1.0 or 1.1 running on the target machine
- RASM-86 , the Digital Research Relocatable Assembler, or an assembler producing Intel Object Module Format
- LINK-86 , the Digital Research Linker/Locator, or a linker that accepts Intel Object Module Format and produces CMD format files
- DDT-86 or SID-86 , Digital Research "debuggers", or another debugger
- And ideally, a second CRT device connected to a serial port on the target machine

Examples in this manual use RASM-86, LINK-86, and DDT-86.

This manual assumes extensive knowledge of the 8086/8088 microprocessors, and also an understanding of the target machine's hardware. You should be familiar with the following manuals, which together with this manual, document CP/M-86 Plus:

- The CP/M-86 Plus Operating System User's Guide (hereinafter cited as the User's Guide) describes the user's interface to CP/M-86 Plus, how to use the Digital Research utility programs supplied with CP/M-86 Plus, and other features used to execute application programs.

- The CP/M-86 Plus Operating System Programmer's Reference Guide (hereinafter cited as the Programmer's Guide) explains the operating system for use by the application programmer. The Programmer's Guide discusses system calls and DDT-86.

- The Programmer's Utilities Guide for the CP/M-86 Family of Operating Systems (hereinafter cited as the Programmer's Utilities Guide) documents the Digital Research utility programs, RASM-86 and LINK-86, used to assemble and link software written for CP/M-86 Plus.

Digital Research does not support any additions or modifications made to CP/M-86 Plus by the OEM or distributor. Any BIOS or utility that is written or modified by the OEM must be supported by the OEM.

The following are terms, conventions, and abbreviations used in this manual:

- CP/M-86 1.X refers to either CP/M-86 1.0 or CP/M-86 1.1.

- The term "process" is used synonomously with "program" and refers to the environment a program executes in, distinct from other concurrently running programs.

- Initial letters of names of all data structures internal to the operating system or BIOS are capitalized and are used to form acronyms; for example, DPB is short for Disk Parameter Block.

- The term "system calls" refers to the functions available to application programs and performed by the operating system. These calls, shown in Appendix E, have mnemonic names and appear in all capital letters.

- The names of the BIOS functions invoked by the Basic Disk Operating System (BDOS) are also mnemonics shown in all capital letters. Each is prefixed with "IO_". Appendix F shows the BIOS functions.

- Variable and label names in the BIOS appear as all capitals, for instance, the BIOSINIT and BIOSENTRY labels.

- The names of utilities, such as RASM-86, GENCPM, and DEVICE, appear in all uppercase.

- 8086/8088 instructions are in all capital letters using the mnemonics recognized by RASM-86. They are followed by the instruction name in parenthesis. The phrase "then executes a CALLF (Call Far instruction) ..." shows this convention.

- 8086/8088 memory references are in a segment:offset format; for example, F000:FFFFh is the last byte in the 8086/8088 megabyte address space.

- Paragraph address or segment address refers to memory locations on even 16-byte boundaries.

- All numbers are decimal values unless suffixed by an "h" denoting hexidecimal (base 16) or a "b" denoting bits. However, the default base for DDT-86 and SID-86 is hexidecimal.

- An "@" prefixes public variables in the BIOS.

- A "?" prefixes public labels in the BIOS.

Table of Contents

Section 1

Introduction to CP/M-86 Plus

This section provides introductory and background material relevant to system implementation. It explains the modules of CP/M-86 Plus, communication between the modules, the hardware CP/M-86 Plus supports, and the features of CP/M-86 Plus. The Programmer's Guide provides a more general overview and explanations of the CP/M-86 Plus system calls.

CP/M-86 PLUS MODULES

The memory resident part of CP/M-86 Plus consists of the following four modules: the Basic Disk Operating System (BDOS), the Basic Input/Output System (BIOS), the System Data Area (SYSDAT), and optionally, the Console Command Processor (CCP).

Figure 1-1 illustrates the layout of CP/M-86 Plus in memory:

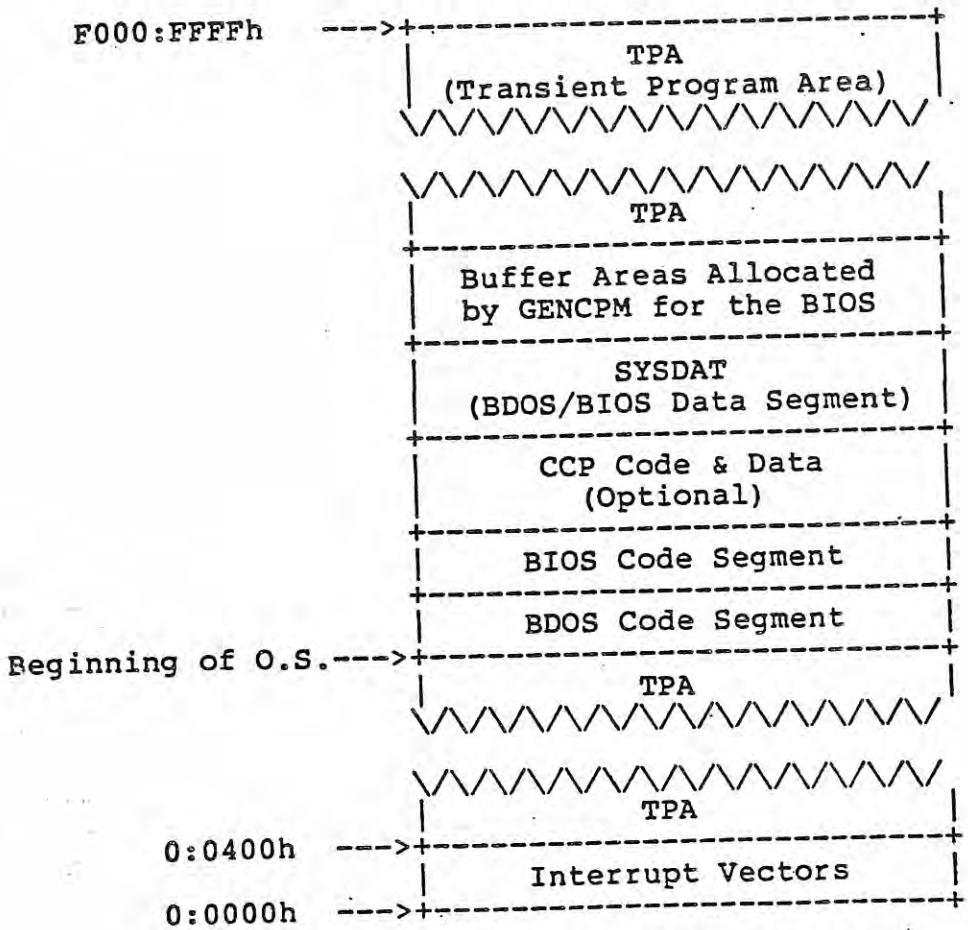


Figure 1-1. General Memory Organization of CP/M-86 Plus

The BDOS is the invarient and logical nucleus of CP/M-86 Plus. This nucleus consists of six sub-modules, one for each functional area of the operating system. These are the six functional areas:

- entry to and exit from of the operating system
- loading transient programs from disk
- performing character I/O
- performing file I/O
- allocating and freeing memory
- scheduling and managing processes

The BIOS allows CP/M-86 Plus to run on a specific computer. It consists of an invarient BIOS Kernel and a set of hardware-dependent modules that interface to the Kernel.

The CCP provides the basic user interface to the facilities of the operating system and supplies six commands: DIR, DIRS, ERASE, RENAME, TYPE, and USER. These internal commands are part of the CCP; other commands are called transient programs (applications) and are disk resident. Transient programs load into memory, execute, then return control to the CCP. The User's Guide documents the operation of the CCP.

The CCP can be made a part of the operating system memory image or it can be loaded and run as a transient program. GENCPM, the system generation utility, controls this option. Figure 1-1 shows the CCP as a permanent part of the operating system.

When the CCP is not part of the operating system image, it must be available through the drive search chain. The drive search chain allows up to four drives to be searched for a disk resident command or submit file. (See the SETDEF utility in the User's Guide).

The BDOS and the BIOS modules cooperate to provide the CCP and other transient programs with a set of hardware independent operating system functions. Because the BIOS is configured for different hardware environments and the BDOS remains constant, you can transfer programs that run under CP/M-86 Plus unchanged to systems with different hardware configurations.

The System Data Area (SYSDAT) is the data segment for the BDOS and BIOS. It contains system variables, including values set by GENCPM, pointers to the system tables, and most of the data structures used by the BDOS and the BIOS.

The BIOS must be separate code and data, with all stack and extra segments included in the data. The BIOS Data begins at location 0F00h relative to the beginning of the SYSDAT segment. Appendix C shows the format of SYSDAT. The S_SYSVAR system call, documented in the Programmer's Guide, allows some SYSDAT Data and other internal BDOS data fields to be queried and changed by transient programs.

MODULE COMMUNICATION

After the system initialization, the BDOS passes control to the CCP. If the CCP is not part of the operating system image, it loads from disk and runs as a transient program. The CCP prompts for commands and if required, requests disk-based transients be loaded and run by the BDOS through the P_CHAIN system call.

Transients communicate with CP/M-86 Plus through system calls. Appendix E lists these system calls; the Programmer's Guide documents them. The BDOS implements all system calls.

The BDOS calls the BIOS to perform hardware-dependent functions, requesting a specific function and passing parameters using a set of register conventions.

HARDWARE ENVIRONMENT

You can customize the BIOS to match any hardware environment with the following characteristics:

- Intel 8086 or 8088
- 128 Kbytes up to 1-megabyte of Random Access Memory (RAM)
- 1 to 16 logical drives, each with up to 512 megabytes, formatted capacity
- 1 to 16 character I/O devices; one of which must be the system console. Other possible character devices are printers, plotters, and communications hardware.

CP/M-86 Plus without the CCP or BIOS occupies about 21 Kbytes of memory. A minimum of 128 Kbytes of RAM is recommended for the operation of CP/M-86 Plus when you use it for general purposes with a variety of application software. In a dedicated application, CP/M-86 Plus RAM requirements can be just the operating system, the BIOS, and the application.

Memory does not need to be contiguous other than that occupied by the operating system image. The Transient Program Area (TPA) is the memory available for disk-based programs. Several discontiguous memory regions, as shown in Figure 1-1, can make up the TPA. These regions need not be contiguous with the operating system. Transient programs, however, require contiguous memory large enough for all the relocatable groups specified in the transient's CMD header since one memory allocation is made for these groups. The Programmer's Guide discusses the CMD format.

CP/M-86 Plus is usually a disk-based system. However, other mass storage devices such as ROMs, cassette tape, and bubble memory can be made to appear as disk drives, storing the operating system image

and transient programs. An example of this is using part of RAM to act as a disk drive, resulting in a high speed temporary disk.

If CP/M-86 Plus is placed in ROM, sufficient RAM must exist for the operating system data area. Appendix G discusses putting CP/M-86 Plus in ROM.

If an 8087 numeric processor is present, only one process can use it at a time. A program needing the 8087 can not load if another program is currently using the 8087. The BIOS informs the operating system at initialization time if an 8087 is present.

FEATURES OF CP/M-86 PLUS

CP/M-86 Plus includes many new features representing a major improvement over CP/M-86 1.X. The following list describes those new features and improvements pertinent to customizing CP/M-86 Plus.

- Disk performance, especially random I/O, is improved over that of CP/M-86 1.X by hash coding the directory entries and Least Recently Used (LRU) buffering for directory and file data.

- The BDOS performs auto-login of removable media drives. The implementation of door open interrupts on removable media drives is highly recommended as an additional check on data integrity and for providing disk I/O performance improvements of up to 30%. The door open interrupt information allows the BDOS to treat removable media drives in a manner similar to permanent media drives.

- The file system capacity is larger, allowing a storage capacity of up to 512 megabytes for each of the 16 possible logical drives, and the maximum file size is now 32 megabytes. The file system also provides time and date stamping.

- Live control characters and type-ahead are supported by cooperating routines in the BDOS and BIOS. The live control characters are CTRL-C, CTRL-S, CTRL-Q, and CTRL-P and their functions are performed when a keyboard interrupt occurs.

- As noted before, the CCP can be a permanent part of the system or loaded as a transient program. When the CCP is a transient, it is loaded and remains in memory until the memory is needed by another transient.

- The mapping of the logical devices CONIN:, CONOUT:, AUXIN:, AUXOUT:, and LST: onto different physical devices has been standardized and made more flexible. This allows the dynamic remapping of the console to another device, such as a graphics console. Logical device output can be directed to several physical devices at once. See the DEVICE utility in the User's Guide.

- CP/M-86 Plus can run up to four programs (processes) at once, one in the foreground and up to three in the background. Only the foreground program has access to the physical console; the background programs must have console I/O redirected from and to files.

- The use of RASM-86 and LINK-86 to assemble and link the system modules simplifies support of different hardware configurations and allows the field installation of new drivers.

- The interface to hardware drivers has been simplified and improved by use of a BIOS Kernel. The Kernel is intended for unchanged use in any BIOS implementation. However, the source is provided if you need to alter the Kernel.

- The BDOS now performs Blocking/Deblocking instead of the BIOS. BIOS disk reads and writes transfer physical sectors and up to 16 Kbytes on each call.

- GENCPM creates the CP/M-86 Plus image contained in the CPM3.SYS file and provides many configuration options. GENCPM can automatically all allocates buffers while building the system image. This allows the testing of many combinations of disk and directory buffers enabling the system implementor to optimize disk performance and memory usage.

End of Section 1

Section 2

Customizing CP/M-86 Plus

This section describes the modules of the BIOS that you supply and gives a summary of the tasks involved in porting CP/M-86 Plus.

BIOS MODULES

CP/M-86 Plus introduces the use of a BIOS Kernel to standardize and simplify the porting process. The customization procedure for CP/M-86 Plus entails writing hardware-dependent drivers that interface with the BIOS Kernel. The Kernel is a set of routines and data common to any CP/M-86 Plus BIOS. Porting earlier operating systems in the CP/M family entailed writing the entire BIOS that interfaced directly to the Basic Disk Operating System (BDOS).

The Kernel, and the hardware-dependent modules you supply, are linked together to form the file BIOS3.SYS. GENCPM uses the files BIOS3.SYS, BDOS3.SYS, and optionally, CCP.CMD as input to create the memory image file CPM3.SYS.

Table 2-1 summarizes the hardware-dependent modules you must write, and lists the section where each is discussed in detail.

Table 2-1. OEM-written BIOS Modules

Module	Description
INIT	Initializes all I/O device hardware and prints the BIOS sign-on message. (Section 5)
CHARIO	Performs character I/O for console and other character devices such as printers and communications ports. (Section 6)
DISKIO	Performs disk I/O, media density selection and handling of removable media door open interrupts. (Section 7)
CLOCK	Updates the time of day variables and forces dispatches when more than one program is running. (Section 8)

Examples of these modules are on the distribution diskette. These example modules implement a BIOS for operation on a CompuPro 8/16 with at least 64 Kbytes of RAM. The example modules are in the files INIT.A86, CHARIO.A86, DISKIO.A86, and CLOCK.A86.

Appendix B contains a listing of the BIOS Kernel. The Kernel is also on the distribution diskette in the file BIOSKRNL.A86.

All the BIOS functions, data structures, fields, public variables, and public subroutines of the BIOS Kernel are indexed and cross-referenced for easier access to essential information during design and coding.

CP/M-86 PLUS CUSTOMIZATION TASKS

The entire customization process generally includes the following tasks:

- Prepare the customized hardware-dependent BIOS modules, either by modifying the example BIOS modules, by modifying an existing CP/M-86 1.X BIOS, or by expanding the loader BIOS if you have written it first.
- Test and debug the hardware-dependent BIOS modules. Wherever possible, debug these modules as transient programs under a running CP/M-86 1.X.
- Process all the OBJ BIOS modules through the MODEDIT utility in order to resolve external references.
- Create the BIOS3.SYS file by linking the modules together using LINK-86.
- Build the CP/M-86 Plus system image using the GENCPM utility. GENCPM initializes values in the system image according to information given by the system programmer.
- Debug the BIOS under CP/M-86 1.X using a remote console.
- Prepare, test, and debug the CP/M-86 Plus bootstrap loader using an existing CP/M-86 1.X bootstrap loader as a base.
- Prepare the CP/M-86 Plus loader BIOS and integrate with the loader BDOS to create the CP/M-86 Plus loader, CPMLDR.
- Write the bootstrap and loader onto the system tracks of a boot diskette using the TCOPY program.
- Boot CP/M-86 Plus from disk using CPMLDR and test.
- Write, test, and debug any hardware-dependent utilities, such as a disk formatter.

The preceding list presents a formidable task, especially if you are unfamiliar with porting Digital Research operating systems. Appendix A outlines a series of base levels or steps to help organize and simplify the creation of a new BIOS.

End of Section 2

Section 3

BIOS Kernel

This section describes how the BIOS Kernel interfaces with the BIOS modules you supply and how the BIOS Kernel interfaces with the BDOS. With the exception of interrupt service routines, all communication between the BDOS and the hardware drivers contained in the INIT, CHARIO, DISKIO, and CLOCK modules occurs through the BIOS Kernel.

The BIOS Kernel and the example BIOS modules prefix public labels with a "?" and public data variables with an "@". Appendix B lists the BIOS Kernel for reference in reading this and subsequent sections.

The BIOS Kernel is intended to be used unchanged in your implementation of CP/M-86 Plus. Though you can modify the Kernel if necessary, the Kernel Data and Code Headers as defined in this section must be present in any BIOS.

BIOS KERNEL DATA HEADER

The BIOS data begins with the BIOS Kernel Data Header at offset 0F00h relative to the SYSDAT segment value. The BIOS data segment is the same as the SYSDAT segment and the BDOS sets the DS register to the SYSDAT segment, before calling the BIOS.

The BDOS, the BIOS Kernel, and the other BIOS modules access the Data Header. Since the BIOS Data Header is in a fixed format and location, access to hardware-dependent information independent of a particular BIOS implementation is possible. For instance, GENCPM and DEVICE are two utilities that rely on the Data Header for information about character and disk devices.

The following code fragment from the BIOS Kernel shows the layout of the BIOS Kernel Data Header.

Variables in the BIOS Kernel Data Header are prefixed with "BH_" to help identify them. The @CDBA-@CDBP and the @DPHA-@DPHP variables in the header do not have this prefix since they are external symbols in the BIOS Kernel and are defined in the DISKIO and CHARIO modules.

Listing 3-1. BIOS Kernal Data Header

```
;*********************************************************
;
;       BIOS Kernel Data Header
;
;*********************************************************
        org     0000h
                ;use the LINK-86 [data[origin[0F00]]] option
                ;to set the origin of the data segment at 0F00h

@bh_delay       db      0               ;0FFh if process delaying
@bh_ticksec     db      60              ;ticks per second
@bh_gdopen      db      0               ;0FFh if drive door opened
@bh_inint       db      0               ;in interrupt count
@bh_nextflag    db      4               ;next available flag
@bh_lastflag    db      0               ;last available flag
@bh_intconin    db      0               ;0FFh if interrupt driven CONIN:
@bh_8087        db      0               ;0FFh if 8087 exists

;       disk parameter header offset table

@bh_dphtable    dw      offset @dpha    ;drive A:
                dw      offset @dphb    ;drive B:
                dw      offset @dphc    ;drive C:
                dw      offset @dphd    ;drive D:
                dw      offset @dphe    ;drive E:
                dw      offset @dphf    ;drive F:
                dw      offset @dphg    ;drive G:
                dw      offset @dphh    ;drive H:
                dw      offset @dphi    ;drive I:
                dw      offset @dphj    ;drive J:
                dw      offset @dphk    ;drive K:
                dw      offset @dphl    ;drive L:
                dw      offset @dphm    ;drive M:
                dw      offset @dphn    ;drive N:
                dw      offset @dpho    ;drive O:
                dw      offset @dphp    ;drive P:

;       character device block offset table

@bh_cdbtable    dw      offset @cdba    ;device A
                dw      offset @cdbb    ;device B
                dw      offset @cdbc    ;device C
                dw      offset @cdbd    ;device D
                dw      offset @cdbe    ;device E
                dw      offset @cdbf    ;device F
                dw      offset @cdbg    ;device G
                dw      offset @cdbh    ;device H
                dw      offset @cdbi    ;device I
                dw      offset @cdbj    ;device J
                dw      offset @cdbk    ;device K
                dw      offset @cdbl    ;device L
                dw      offset @cdbm    ;device M
```

```
                dw      offset @cdbn    ;device N
                dw      offset @cdbo    ;device O
                dw      offset @cdbp    ;device P

;       Character device roots for console input,
;       console ouput, auxiliary input, auxiliary output
;       and list output.

@bh_ciroot      dw      offset @cdba    ;console input
@bh_coroot      dw      offset @cdba    ;console output
@bh_airoot      dw      offset @cdbb    ;aux input
@bh_aoroot      dw      offset @cdbb    ;aux output
@bh_loroot      dw      offset @cdbc    ;list output

@bh_bufbase     dw      0               ;offset of buffer
@bh_buflen      dw      0               ;length of buffer

@bh_memdesc     rw      32*3            ;room for 32 memory descriptors

@bh_chain       dw      chain_msg       ;chain error message address
@bh_prompt      dw      prompt_msg      ;error CCP prompt message address
@bh_user        dw      user_str        ;error CCP USER command string
```

Table 3-1 describes each field in the Data Header. The offset for each data field is in parentheses next to the field name.

Table 3-1. BIOS Data Header Fields

Field	Explanation
@BH_DELAY (0F00h)	When a program makes a P_DELAY system call, this field is set to 0FFh by the BDOS. When @BH_DELAY is equal to 0FFh, the tick interrupt service routine in the BIOS CLOCK module sets system flag number 1 (the tick flag) on every system tick. System flags are set through the INT_SETFLAG function described in Section 4 of this guide. The BDOS in turn decrements the tick count for delaying processes on each INT_SETFLAG operation made to flag number 1. When no processes are delaying, @BH_DELAY is set to 0 by the BDOS and the CLOCK module does not set the tick flag. @BH_DELAY is initialized to 0 in the BIOS Kernel Data Header.

Table 3-1. (continued)

Field Explanation

@BH_TICKSEC (0F01h)

The number of system ticks per second. This field is initialized by GENCPM, but can be modified by the your ?CLOCK_INIT routine. A transient program can read this variable using the S_SYSVAR system call and calculate the number of ticks needed for a P_DELAY system call. The tick interrupt service routine also forces dispatches between CPU bound processes on each system tick. Setting @BH_TICKSEC to 0 signifies ticks are not supported by the BIOS and the BDOS does not allow P_DELAY system calls nor support multitasking. A typical setting for this field is 60, specifying a system tick every 16.66 milliseconds.

@BH_GDOPEN (0F02h)

[Global door open] This field is set to 0FFh by the drive door open interrupt service routine when any disk drive door has been opened. The BDOS checks this field before every disk operation to verify the media has not changed. The door open interrupt service routine must also set the DPH_MEDIAFLAG in the Disk Parameter Header (DPH) associated with the drive. The DPH_MEDIAFLAG specifies to the BDOS which drives had doors opened.

@BH_GDOPEN is initialized to 0 in the BIOS Kernel Data Header.

@BH_ININT (0F03h)

[In interrupt count] This field is incremented upon entry to and decremented prior to exiting from an interrupt service routine. @BH_ININT counts the number of interrupt service routines currently being executed. This count can become greater than one when an interrupt service routine reenables interrupts, thereby allowing another interrupt to occur. Keeping track of the number of interrupts being serviced prevents waiting for an entire system tick before finishing an interrupt service routine. "Interrupt Device Drivers" in Section 4 discusses the use of @BH_ININT.

If interrupts are not reenabled within any of the interrupt service routines, @BH_ININT does not need to be incremented or decremented. This field is initialized to 0 in the BIOS Kernel Data Header.

Table 3-1. (continued)

Field	Explanation

@BH_NEXTFLAG (0F04h)

This is the next system flag available for allocation to an interrupt routine. Device drivers that need a system flag for ?WAITFLAG and INT_SETFLAG operations use this field to obtain a specific flag number to use. Flags are numbered from zero and the first four are reserved for use by the BDOS. Thus, GENCPM initializes @BH_NEXTFLAG to 4.

@BH_LASTFLAG (0F05h)

The value in this field indicates the last system flag available for system use. This is the number of flags in the system minus one. GENCPM sets this field according to the number of flags requested by BIOS data structures and the additional number of flags you request when running GENCPM.

The BIOS must ensure @BH_NEXTFLAG is less than or equal to @BH_LASTFLAG before allocating a flag. "Device Initialization" in Section 5 discusses the allocation of flags at initialization time.

@BH_INTCONIN (0F06h)

[Interrupt console input] When set to OFFh, this field indicates that the CONIN: device (the logical console input device) is interrupt-driven. If this field is 0FFh, the BDOS does not call the IO_CONST function in the BIOS Kernel when a transient makes console output or console input system calls. If @BH_INTCONIN is set to 0FFh, the interrupt service routine associated with the current CONIN: device must call the BDOS INT_CHARSCAN routine so the BDOS can scan for CTRL-S, CTLR-Q, CTRL-C, and CTRL-P.

The current CONIN: device is specified by the Character Device Block (CDB), which is pointed to by the console input root (@BH_CIROOT in this table). A field in the Character Device Block (discussed in Section 6) indicates whether input from the device is interrupt-driven.

The Kernel BIOSINIT routine initializes @BH_INTCONIN and when the DEVICE utility changes the @BH_CIROOT, it also updates @BH_INTCONIN.

Table 3-1. (continued)

Field	Explanation

@BH_8087 (0F07h)

This field is set to 0FFh by your INIT module if the 8087 is present and to 0 if is not. Transient programs are marked as 8087 users by a field in the file CMD header. The BDOS successfully loads transients needing the 8087 only if @BH_8087 is set to 0FFh. Additionally, the BDOS permits only one process to use the 8087 at a time because the 8087 registers are not saved when two or more processes are running simultaneously.

@DPHA-@DPHP (0F08h)

This is the table of offsets of Disk Parameter Headers (DPHs) for logical drives A through P respectively. The DPHs are declared as externals in the BIOS Kernel and are publics defined in the DISKIO modules. (DISKIO modules refer to all the modules you supply containing disk drivers.) GENCPM uses these offsets to find DPHs and to build any requested data and disk buffers, checksum and allocation vectors, and hash tables.

The DPH is a data structure used by the file system for performing disk I/O on a particular logical drive. The DPH contains the offsets for drive initialization, drive login, drive read, and drive write routines as well as the offset to the Disk Parameter Block (DPB). The DPB defines the characteristics of a physical drive. Section 7 discusses the DPB and DPH in detail.

LINK-86 sets each DPH field in this table to the offset of the corresponding DPH defined in the DISKIO modules or to 0 if the DPH is not defined.

@CDBA-@CDBP (0F28h)

This is the table of offsets of Character Device Blocks (CDB) for devices A through P respectively. The CDBs are declared as externals in the BIOS Kernel and are publics defined in the CHARIO modules. (CHARIO modules refer to all the modules you supply containing character device drivers.)

Table 3-1. (continued)

Field	Explanation
@CDBA-@CDBP (continued)	CP/M-86 Plus supports a maximum of 16 character devices, each of which is described by a CDB. The CDBs contain an ASCII device name and offsets of device initialization, device input, device input status, device output, and device output status routines. The CDB also contains information on baud and protocol the device is currently programmed to support, as well as other protocols it can potentially support. Section 6 discusses CDBs in detail. The DEVICE utility uses the CDB offsets in the Kernel Data Header to change the mapping of logical character devices to physical devices and to dynamically change baud and protocol configurations. LINK-86 sets each CDB field in this table to the offset of the corresponding CDB defined in the CHARIO modules or to 0 if the CDB is not defined.
@BH_CIROOT (0F48h)	This is the offset of the Character Device Block (CDB) currently attached to the logical console input device CONIN:. Console input comes from the device associated with this CDB. The system implementor initializes this field with CDB symbol (@CDBA-@CDBP) of the initial CONIN: device. This CDB external is resolved by LINK-86 and must result in a non-zero value in @BH_CIROOT.
@BH_COROOT (0F4Ah)	This is the list of Character Device Blocks (CDBs) currently attached to the logical console output device CONOUT:. @BH_COROOT contains the offset of the first CDB on this list. Each character output to CONOUT: is sent to each of the physical devices represented by the CDBs on this list. The system implementor initializes this field with the CDB symbol (@CDBA-@CDBP) of the initial CONOUT: device. This CDB external is resolved by LINK-86 and must result in a non-zero value in @BH_COROOT.

Table 3-1. (continued)

Field Explanation

@BH_AIROOT (0F4Ch)

This is the offset of the Character Device Block (CDB) currently attached to the logical auxiliary input device AUXIN:. Auxiliary device input comes from the device associated with this CDB. The system implementor initializes this field with the CDB symbol (@CDBA-@CDBP) of the initial AUXIN: device. This CDB external is resolved by LINK-86.

@BH_AOROOT (0F4Eh)

This is the list of Character Device Blocks (CDBs) currently attached to the logical auxiliary output device AUXOUT:. @BH_AOROOT contains the offset of the first CDB on this list. Each character output to AUXOUT: is sent to each of the physical devices represented by the CDBs on this list. The system implementor initializes this field with the CDB symbol (@CDBA-@CDBP) of the initial AUXOUT: device. This CDB external is resolved by LINK-86.

@BH_LOROOT (0F50h)

This is the list of Character Device Blocks (CDBs) currently attached to the logical list device LST:. @BH_LOROOT contains the offset of the first CDB on this list. Each character output to the LST: is sent to each physical device represented by the CDBs on this list. The system implementor initializes this field with the CDB symbol (@CDBA-@CDBP) of the initial LST: device. This CDB external is resolved by LINK-86.

@BH_BUFBASE (0F52h)

This is the offset of the uninitialized buffer in the SYSDAT segment for use by the BIOS. GENCPM sets this field and reserves the buffer in the CP/M-86 Plus system image. Section 9 discusses GENCPM.

@BH_BUFLEN (0F54h)

This is the size, in paragraphs, of the uninitialized buffer in the SYSDAT segment optionally created by GENCPM.

Table 3-1. (continued)

Field Explanation

@BH_MEMDESC (0F56h)

This is the table of 32 Memory Descriptors, which are each 6 bytes long. GENCPM initializes this table when you answer the GENCPM memory definition questions. Appendix G shows and discusses the Memory Descriptor format.

@BH_CHAIN (1016h)

This is the offset of the error message used by the BDOS P_CHAIN system call when an error is encountered after the calling process has released its memory. The offset in @BH_CHAIN must be defined and address a printable string terminated by a '$'. The default string defined in the BIOS Kernel is:

```
chain_msg   db      13,10,'Cannot Load Program',13,10,'$'
```

0:6444

This string can be changed to a foreign language message, though the CRLF sequences (13,10) should be kept. Appendix H discusses foreign error message customization.

@BH_PROMPT (1018h)

This is the offset of the prompt used by the Error CCP when the CCP is not a permanent part of the system and the CCP.CMD file cannot be found on disk. (The Error CCP is described in the User's Guide.) @BH_PROMPT must be defined and address a printable string terminated by a '$'. The default string defined in the BIOS Kernel is:

```
prompt_msg  db      13,10,'Cannot Load CCP $'
```

This string can be changed to a foreign language message, though the prefixed CRLF sequence (13,10) should be kept. See Appendix H.

@BH_USER (101Ah)

This is the offset of the string used by the Error CCP to recognize the one internal command performed by the Error CCP. @BH_USER must be defined and address a byte followed by the message string. The first byte is the number of characters in the following string. The default string defined in the BIOS Kernel is:

```
user_str    db      4,'USER'
```

This string can be changed to a foreign language as required. See Appendix H.

BDOS/BIOS INTERFACE

The BDOS calls the BIOS through two entry points in the BIOS Kernel. All communication to the BIOS is performed through these points.

BIOS Kernel Code Header

The BIOS Kernel Code Header is located at offset 0 relative the BIOS code segment. It consists of jumps to BIOSINIT and BIOSENTRY, and the SYSDAT segment address. The BDOS performs a single CALLF (Call Far instruction) to JMP BIOSINIT after system boot. Each time the BDOS must have access to the hardware it performs a CALLF to JMP BIOSENTRY. The double word pointers the BDOS uses to find these two entries reside at 2Ch and 28H in SYSDAT. (Appendix C shows the SYSDAT format.)

The SYSDAT segment value, which is also the BIOS data segment is kept in the code segment of the BIOS to be accessible from interrupt service routine.

The Code Header is in the following fragment from the BIOS Kernel:

Listing 3-2. BIOS Kernel Code Header

```
;*****************************************************
;
;  BIOS CODE HEADER
;
;*****************************************************

CSEG
   org     0000h

   jmp     biosinit                ;BIOS initialization entry
   jmp     biosentry               ;BIOS function entry

@sysdat            rw      1              ;OS Data Segment
```

Section 5 discusses the BIOSINIT routine in BIOS initialization.

BIOSENTRY Routine

The Kernel BIOSENTRY routine receives from the BDOS, a BIOS function

number in AL, and parameters in CX and DX or on the stack as needed. Fifteen levels of stack are available to the BIOS when the BDOS calls BIOSENTRY. The value in AL indexes into the BIOS function table, which is located in the BIOS Kernel. Before calling BIOSENTRY, the BDOS sets DS to SYSDAT and ES to the currently running process environment. DS and ES must be preserved through the Kernel and the routines in the other BIOS modules. The first comment in Listing 3-3 summarizes the BDOS/BIOS register conventions.

The S_BIOS system call in the BDOS does not range check for BIOS functions 80h and above to allow BIOS functions specific to your CP/M-86 Plus implementation. The example BIOS supports no functions above 80h and these functions return errors as shown in the following BIOSENTRY routine:

Listing 3-3. Kernel BIOSENTRY Routine

```
CSEG

;*********************************************************
;
;               BIOS ENTRY
;
;*********************************************************

;=========
biosentry:      ; BIOS Entry Point
;=========
; All calls to the BIOS after INIT, enter through this code
;  with a CALLF and must return with a RETF.
;       Entry:  AL = function number
;               CX = first parameter
;               DX = second parameter
;               DS = system data segment
;               ES = process environment (preserved through call)
;       Exit:   AX = BX = return or BIOS error code
;               DS = SYSDAT segment
;               ES = process environment (preserved through call)
;               SS,SP must also be preserved
;               CX,DX,SI,DI,BP can be changed by the BIOS
```

Listing 3-3. (continued)

```
        cmp al,80h ! jae range_er       ;check for BIOS functions
                                        ;above 80h
          cld                           ;clear direction flag
          xor ah,ah ! shl ax,1          ;index into BIOS function
                                        ;table
          mov bx,ax
          call functab[bx]              ;call BIOS kernel routine
          mov es,rlr                    ;restore ES
bdos_ret:
          mov bx,ax                     ;BX = AX
          retf
range_err:
        mov ax,0FFFFh                   ;function out of range
        jmps bdos_ret

DSEG

functab         dw      io_conist       ; 0 - console status
                dw      io_conin        ; 1 - console input
                dw      io_conout       ; 2 - console output
                dw      io_listst       ; 3 - list output status
                dw      io_list         ; 4 - list output
                dw      io_auxin        ; 5 - aux input
                dw      io_auxout       ; 6 - aux output
                dw      io_notimp       ; 7 - CCP/M function
                dw      io_notimp       ; 8 - CCP/M function
                dw      io_seldsk       ; 9 - select disk
                dw      io_read         ;10 - read sector
                dw      io_write        ;11 - write sector
                dw      io_flush        ;12 - flush buffers
                dw      io_notimp       ;13 - CCP/M function
                dw      io_devinit      ;14 - char. device init
                dw      io_conost       ;15 - console output status
                dw      io_auxist       ;16 - aux input status
                dw      io_auxost       ;17 - aux output status
```

The BIOS Kernel assumes the other BIOS modules preserve DS and ES. DS is the SYSDAT segment value and the data segment of the BIOS. ES is the currently running process environment. If you change DS or ES, save them using PUSH and POP instructions. Alternatively, SYSDAT is always available through the Kernel @SYSDAT public defined in the code segment, and the currently running process is kept at location 4Eh in the SYSDAT segment. Location 4Eh in SYSDAT is the Ready List Root as shown in Appendix C.

BIOS Kernel Functions Called by the BDOS

The BDOS calls the BIOS Kernel through the BIOSENTRY routine to perform any hardware-dependent actions. The BIOS functions used by the BDOS fall into two groups: character I/O and disk I/O. BIOS function numbers 7,8, and 13 are reserved for compatibility with Concurrent CP/M..and return an 0FFFFh in AX and BX from the CP/M-86 Plus BIOS. All BIOS functions called by the BDOS begin with the prefix "IO_" and are the functions in the "FUNCTAB" in Listing 3-3 earlier. The following table shows the two groupings of BIOS functions available to the BDOS:

Table 3-3. BIOS Kernel IO_ Functions

No.	Mnemonic	Meaning
	Character Device I/O Functions	
0	IO_CONIST	CONSOLE INPUT STATUS
1	IO_CONIN	CONSOLE INPUT
2	IO_CONOUT	CONSOLE OUTPUT
3	IO_LISTST	LIST STATUS
4	IO_LISTOUT	LIST OUTPUT
5	IO_AUXIN	AUXILIARY INPUT
6	IO_AUXOUT	AUXILIARY OUTPUT
14	IO_DEVINIT	DEVICE INITIALIZATION
15	IO_CONOST	CONSOLE OUTPUT STATUS
16	IO_AUXIST	AUXILIARY INPUT STATUS
17	IO_AUXOST	AUXILIARY OUTPUT STATUS
	Disk I/O Functions	
9	IO_SELDSK	SELECT DISK
10	IO_READ	READ DISK
11	IO_WRITE	WRITE DISK
12	IO_FLUSH	FLUSH BUFFERS

BIOS KERNEL/BIOS MODULES INTERFACE

All IO_ functions are in the BIOS Kernel. Most of these functions use the the Character Device Blocks (CDBs) and the Disk Parameter Headers (DPHs) to locate hardware-dependent routines within the other BIOS modules. The BDOS reserves fifteen levels of stack to be used by the BIOS Kernel on each call to the BIOSENTRY routine. This is extra stack area past any parameters passed to the BIOS on the stack. The IO_ functions use differing amounts of stack space before calling the hardware-dependent routines you supply in the other BIOS modules. If your routines need more stack space, they must switch to a local stack.

BIOS Kernel/CHARIO Interface

The BIOS Kernel Character IO_ functions serve as a layer between the BDOS and the physical character I/O routines addressed from the CDBs. The BDOS calls the BIOS Kernel functions IO_CONIN, IO_CONIST, IO_CONOUT, IO_CONOST, IO_AUXIN, IO_AUXIST, IO_AUXOUT, IO_AUXOST, IO_LIST, and IO_LISTST to perform character I/O. These character IO_ functions relate the logical CP/M-86 character devices CONIN:, CONOUT:, AUXIN:, AUXOUT:, and LST: to the physical character devices; the character IO_ functions perform the logical to physical mapping of character I/O.

The three logical output devices are mapped onto physical devices by three linked lists of CDBs. The offsets of the first CDB in these lists are contained in the BIOS Kernel Data Header variables @BH_COROOT, @BH_AOROOT, and @BH_LOROOT. The two logical input devices are mapped onto physical devices by the two variables @BH_CIROOT and @BH_AIROOT, which contain the offset of the one CDB associated with the logical device. These offsets in the Data Header are called the character I/O redirection roots:

Table 3-4. Character I/O Redirection Roots

Name	Logical Device
@BH_CIROOT	CONIN: - Console Input
@BH_COROOT	CONOUT: - Console Output
@BH_AIROOT	AUXIN: - Auxiliary Input
@BH_AOROOT	AUXOUT: - Auxiliary Output
@BH_LOROOT	LST: - List Output

Logical device output can go to any combination of up to the sixteen maximum physical character devices. The BIOS Kernel routines IO_CONOUT, IO_AUXOUT, and IO_LIST call the character output routine in each CDB linked to the corresponding device root @BH_COROOT, @BH_AOROOT, or @BH_LOROOT respectively. However, logical device input can be received from only the one physical device since the input device roots, @BH_CIROOT and @BH_AIROOT, are not linked and address only one CDB.

Table 3-5 summarizes the BIOS Kernel character IO_ functions. Note the special handling when a character device root is zero, indicating no physical device is attached to the logical device. The BIOS Kernel listing in Appendix B shows the register conventions for the IO_ functions.

Table 3-5. BIOS Kernel Character IO_ Functions

Function	Definition
IO_CONIN	Calls the CDB_INPUT routine for the CDB addressed by @BH_CIROOT. If @BH_CIROOT is 0, then returns a null (AL=0).
IO_AUXIN	Calls the CDB_INPUT routine for the CDB addressed by @BH_AIROOT. If @BH_AIROOT is 0, then returns a null (AL=0).
IO_CONIST	Calls the CDB_INSTAT routine for the CDB addressed by the @BH_CIROOT. If @BH_CIROOT is 0, then returns not ready status (AL=0).
IO_AUXIST	Calls the CDB_INSTAT routine for the CDB addressed by the @BH_AIROOT. If @BH_AIROOT is 0, then returns not ready status (AL=0).
IO_CONOUT	Calls the CDB_OUTPUT routine for every CDB on the linked list that starts with the CDB addressed by @BH_COROOT. CL is set by the BDOS and is the character to output. IO_CONOUT saves this value and the position in the CDB list between calls to the CDB_OUTPUT routines. If @BH_COROOT is 0, then return.
IO_AUXOUT	Calls the CDB_OUTPUT routine for every CDB on the linked list that starts with the CDB addressed by @BH_AOROOT. CL is set by the BDOS and is the character to output. IO_AUXOUT saves this value and the position in the CDB list between calls to the CDB_OUTPUT routines. If @BH_AOROOT is 0, then returns.
IO_LIST	Calls the CDB_OUTPUT routine for every CDB on the linked list that starts with the CDB addressed by @BH_LOROOT. CL is set by the BDOS and is the character to output. IO_LIST saves this value and the position in the CDB list between calls to the CDB_OUTPUT routines. If @BH_LOROOT is 0, then return.
IO_CONOST	Calls the CDB_OUTSTAT routine for every CDB on the linked list that starts with the CDB addressed by @BH_COROOT. IO_CONOST returns a ready status (AL=0FFh) only if all the devices are ready. If @BH_COROOT is 0, then also return a ready status.

Table 3-5. (continued)

Function	Definition
IO_AUXOST	Calls the CDB_OUTSTAT routine for every CDB on the linked list that starts with the CDB addressed by @BH_AOROOT. IO_AUXOST returns a ready status (AL=0FFh) only if all the devices are ready. If @BH_AOROOT is 0, then also return a ready status.
IO_LISTST	Calls the CDB_OUTSTAT routine for every CDB on the linked list that starts with the CDB addressed by @BH_LOROOT. IO_LISTST returns a ready status (AL=0FFH) only if all the devices are ready. If @BH_LOROOT is 0, then also return a ready status.
IO_DEVINIT	Calls the CDB_INIT routine using the CDB offset in BX. The IO_DEVINIT is available to utilities such as DEVICE through the S_BIOS system call. BX is set by the BDOS before calling IO_DEVINIT. IO_DEVINIT sets register DL to 1 before calling the CDB_INIT routine to indicating that this is not the first initialization call to the device. The Kernel BIOSINIT routine (discussed in Section 5) sets DL to 0 before making the first initialization call to all CDB_INIT routines. Your CDB_INIT routine must return success (AX=0) and failure (AX=0FFFFh) back to the Kernel IO_DEVINIT function and thus back to the BDOS.

Figure 3-1 illustrates character I/O redirection. Console ouput echoes to the printer without use of the CTRL-P command. The @BH_COROOT field in the BIOS Data Header points to the CRT0 CDB, and the CDB_COLINK field within the CRT0 CDB contains the offset of the LPT0 CDB. The IO_CONOUT function in the Kernel calls the console output routine for each device with every character. The addresses of the console ouput routines are contained in the CDB for the respective device. Section 7 defines in detail the CDB structure.

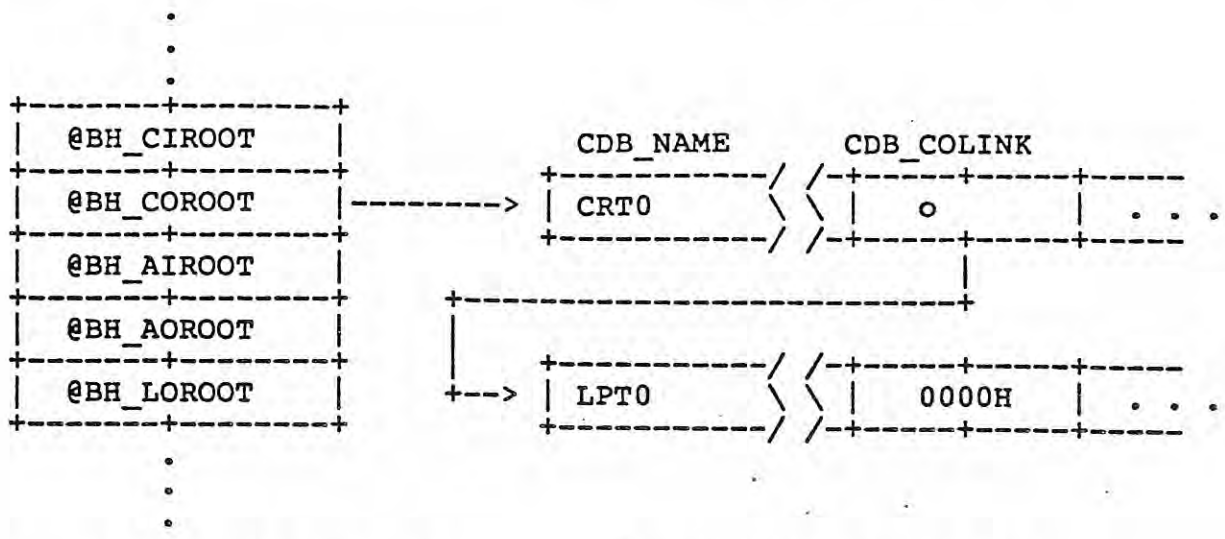


Figure 3-1. Character I/O Redirection Example

BIOS Kernel/BIOS DISKIO Interface

The sixteen DPH offsets in the BIOS Kernel Data Header correspond to the sixteen CP/M-86 Plus logical drives, A:-P:. The DPH structures contain the offsets of the hardware-dependent routines to perform disk I/O. Fields within DPHs are prefixed with the letters "DPH_". The BIOS Kernel listing in Appendix B shows the register conventions for the IO_ functions: in

Table 3-6 defines the BIOS Kernel Disk IO_ functions:

Table 3-6. BIOS Kernel Disk IO_ Functions

Function	Description
IO_SELDSK	The Kernel IO_SELDSK routine indexes into the BIOS Data Header DPH table using the drive requested by the BDOS in register CL. If the DPH field is 0, the drive is not supported by the BIOS, IO_SELDSK returns an error (AX=0). If the BDOS calls IO_SELDSK with the least significant bit (LSB) of register DL set to 0, it is the first time this drive has been selected. On first time selects, IO_SELDSK calls the DPH_LOGIN routine, which can check for media type as discussed in Section 7. If DPH_LOGIN returns successfully IO_SELDSK also returns successfully with the DPH address in AX.

Table 3-6. (continued)

Function	Description
	When IO_SELDSK is called with the least significant bit of DL set, the DPH offset is returned in AX and no call to DPH_LOGIN is made.
IO_READ, IO_WRITE	The Kernel IO_READ and IO_WRITE routines pass all their parameters on the stack. A structure called the I/O Parameter Block (IOPB), which is based on the BP register, is used to access these parameters. The Kernel IO_READ and IO_WRITE routines jump to a common routine that sets up BP, looks up the appropriate DPH, then uses it to call the DPH_READ or DPH_WRITE routine in the DISKIO modules.
IO_FLUSH	This routine is usually not needed since the BDOS reads physical sectors and performs blocking/deblocking to and from logical sectors. If you must perform blocking/deblocking in the BIOS, the IO_FLUSH informs you when "dirty" buffers must be written to disk. The BDOS calls IO_FLUSH when files are closed and program termination occurs. The example BIOS performs no blocking/deblocking and the IO_FLUSH routine simply returns a successful result (AL=0) from the BIOS Kernel.

REENTRANCY IN THE BIOS

BIOS routines do not need to be reentrant. Although several process can be running at the same time, the BDOS allows only one process to call a particular BIOS IO_ function at a time. This does not preclude one process performing disk I/O, another list output, while a third is receiving characters from the keyboard.

A BIOS routine that needs to make CP/M-86 Plus system calls back to the BDOS does so by first setting up the registers exactly like a normal system call, then executing a CALLF (Call Far instruction) to the BDOS double word pointer in SYSDAT as shown in Appendix C. The BIOS Kernel routines ?DISPATCH, ?DELAY, and ?WAITFLAG call the BDOS in this way. The register conventions for these routines are shown in the BIOS Kernel listing. Note that whenever the BDOS is called through the BDOS double word pointer in SYSDAT, the register conventions are the same as for a system call invoked via an INT 224 instruction. (??? ES,DS ???)

When making BDOS calls from the BIOS, you must ensure the BDOS is not calling the same BIOS routine that is making the BDOS call. For

instance, do not make the system call F_WRITE to the BDOS from within the BIOS disk I/O routines, or call C_WRITE when in the device driver currently assigned to CONOUT:. Note that interrupt service routines cannot make system calls to the BDOS. "Interrupt Device Drivers" in Section 4 discusses special BDOS entry points for interrupt service routines.

PUBLIC BIOS KERNEL ROUTINES

Table 3-7 shows the public BIOS Kernel routines that can be used by other BIOS Modules:

Table 3-7. Public BIOS Kernel Routines

Routine	Description
?PMSG	Prints a character string on the current CONOUT: device, using the CDB pointed to by @BH_COROOT. A null byte (0) terminates the string.
?WAITFLAG	Waits for a flag set from the interrupt service routine. A process that must wait for an interrupt to signal completion of a hardware event calls this routine. A flag is allocated using the @BH_NEXTFLAG field in the Kernel Data Header.
?DISPATCH	Gives up the CPU if any other process is ready to run. ?DISPATCH is called by routines polling for hardware status that cannot be interrupt-driven.
?DELAY	Gives up the CPU for the specified number of system ticks. ?DELAY is called by routines that need to wait a specific amount of time when no hardware ready status is available.

End of Section 3

Section 4

Device Drivers

Device drivers are software routines that directly control and communicate with hardware. Usually there is one driver for each physical device. A device driver is actually a collection of several routines to perform initialization and often other I/O functions. For instance, a CP/M-86 Plus console driver refers collectively to the routines for initialization, input, input status, output, and output status. However, a clock driver in CP/M-86 Plus can be simply initialization and an interrupt service routine.

Devices communicate with driver software through the CPU, typically via interrupts or by polling. Interrupts asynchronously signal the CPU when a hardware event occurs. The polling driver, on the other hand, continually interrogates the hardware to determine the occurrence of a hardware event.

This section contrasts interrupt device drivers and polled device drivers in CP/M-86 Plus. Specific information for the console, disk, and clock drivers is in subsequent sections.

INTERRUPT VERSUS POLLED DEVICE DRIVERS

CP/M-86 Plus is designed and optimized for an interrupt-driven BIOS that supplies the operating system a tick every 16 milliseconds (60 times a second). However, a BIOS using polled I/O drivers with no interrupts or tick can also run CP/M-86 Plus.

Interrupt-driven I/O is more efficient than polled I/O. For CP/M-86 Plus, an interrupt-driven console input and a system tick allow the support of type-ahead, live keyboard, and background programs.

Type-ahead lets console input continue independent of what the currently running application program is doing. When the application requests console input, the stored (typed-ahead) characters are sent to the application.

Live keyboard refers to the performance of certain keyboard functions by CP/M-86 Plus independent of what the application program is doing. These functions are the starting (CTRL-S) and stopping (CTRL-Q) of console output, stopping the running process (CTRL-C), and the on and off toggling of printer echo (CTRL-P). Printer echo is the duplication of console output on the printer.

A polled keyboard forces console output to be less efficient than with interrupt keyboard input. When keyboard input is polled, the BDOS must make BIOS IO_CONST calls before each character is output to the console to check for CTRL-S, CTRL-Q, CTRL-C, and CTRL-P.

As mentioned at the end of Section 1, CP/M-86 Plus supports simple multitasking, allowing up to four processes to share the CPU. A system tick forces the rescheduling (dispatching) of the processes currently ready to run. CP/M-86 Plus does not allow the creation of more than one process if a system tick is not supported by the BIOS.

Multitasking is part of CP/M-86 Plus primarily to support printer spooling and plotting, communications, and the ability to monitor other hardware while running a foreground task. Since file protection is not provided in CP/M-86 Plus, multitasking is not a general purpose tool for the end user as it is under Concurrent CP/M.

INTERRUPT DEVICE DRIVERS

A process that needs to wait for a specific interrupt from a hardware device makes a call to the BIOS Kernel ?WAITFLAG routine with the system flag number reserved for the device. The ?WAITFLAG routine either gives up the CPU and waits for the interrupt or returns immediately if the interrupt has already occurred. The interrupt service routine signals the occurence of the hardware event by performing a CALLF (Call Far instruction) to the BDOS INT_SETFLAG routine with the same flag number.

If the physical device causing the interrupt is the current logical CONIN: device, the interrupt service routine performs a CALLF (Call Far instruction) to the BDOS INT_SCANCHAR routine with each character received from the physical device. This physical device is usually the system or a remote console and the INT_SCANCHAR routine allows the BDOS to perform the live keyboard functions.

Interrupt service routines exit by executing a JMPF (Jump Far instruction) to INT_DISPATCH, which is the address of the dispatcher in the BDOS, or by performing an IRET (Interrupt Return instruction). An IRET is executed when other interrupt service routines are "incomplete." In other words, perform an IRET when exiting an interrupt service routine that was invoked while executing a prior interrupt service routine. The interrupt service routines use the BIOS Data Header variable @BH_ININT to signal an interrupt sevice routine in progress. The IRET allows the "incomplete" interrupt service routine from waiting an entire tick (usually 16 milliseconds) or more before it completes. This situation arises when interrupts occur from different devices at almost the same time. It is assumed interrupts do not occur from the same device while executing the interrupt service routine for the device, and thus service routines are not written to be reentrant.

If interrupts are not enabled inside any interrupt service routine in the BIOS, another interrupt cannot preempt the running one and the interrupt can always end with a CALLF to INT_DISPATCH. When the interrupt service routine is short, keeping interrupts off presents no problems. However, if interrupts are off for long periods of time, it can adversely affect applications depending on real-time

response, such as communications packages. The example BIOS reenables interrupts within interrupt service routines to keep interrupt off time to the minimum.

In general, interrupt service routines must follow the steps outlined here. Listing 6-3, 6-4, and 8-1 show example interrupt service routines.

1. Save the DS register by pushing it on the interrupted process"s stack.

2. Set the DS register to @SYSDAT, which is also the BIOS data segment. The following code fragment shows steps 1 and 2:

```
push    ds
mov     ds,@SYSDAT
```

   Since only the value of CS is known upon entry to an interrupt service routine, @SYSDAT is defined within the code segment of the BIOS Kernel, forcing a code segment override.

3. Switch the stack to a local stack. There is no guarantee of the amount of stack space a transient program provides. Provide at least twelve extra stack levels beyond that needed for the interrupt service routine itself. The extra stack is for the BDOS INT_ functions (see Table 4-1) and the occurrence of another interrupt.

4. If any interrupt service routine in your BIOS reenables interrupts on the CPU, increment the @BH_ININT variable. The @BH_ININT must be decremented before the interrupt service routine exits. Interrupts can now be enabled.

5. Save the register environment of the interrupted process, or at least the registers to be used by the interrupt service routine. Usually registers are saved on the local stack established in the previous step.

6. Satisfy the interrupting condition and perform a INT_SETFLAG call to the BIOS Kernel if required. The hardware (usually a PIC, a Programmable Interrupt Controller) should not be reset allowing another interrupt from the same device until interrupts in the 8086/8088 are disabled for the rest of the interrupt service routine, unless the interrupt service routine is reentrant.

7. Restore the register environment of the interrupted process.

8. Switch back to the original stack.

9. Disable interrupts, if they have been enabled, on the CPU for the rest of this interrupt service routine. If this or any of the other interrupt service routines enable interrupts in your BIOS, then decrement the @BH_ININT count. When @BH_ININT

equals 0, no other interrupts are currently being serviced and a JMPF (Jump Far instruction) to the dispatcher can be made. Perform a JMPF to INT_DISPATCH with four words on the stack; the DS register of the interrupted process is followed by the three words pushed by the interrupt. If @BH_ININT is not 0, another interrupt is currently being serviced. In this latter case, POP DS and perform an IRET.

If interrupts are not enabled in any of the interrupt service routines, you can either perform an IRET (Interrupt Return instruction) or a JMPF to INT_DISPATCH. If a CALLF to INT_SETFLAG was performed, it is often desirable for the interrupt service routine to exit by jumping to the dispatcher to awaken the process waiting for the flag set.

Three INT_ functions are the only BDOS routines or functions that can be used from an interrupt service routine. The INT_ functions are only for interrupt service routines and cannot be used from any other part of the BIOS. These functions do not go through the BIOS Kernel for efficiency and to keep interrupt off time to a minimum. All INT_ functions can be invoked with interrupt enabled. The addresses of these functions are in SYSDAT and are double word pointers; Appendix C shows the SYSDAT format. Table 4-1 summarizes the three "INT_" functions and their register conventions.

Table 4-1. BDOS Interrupt Functions

Function	Explanation
INT_SETFLAG	Call Far to this routine to signal the completion of a hardware event. Entry Registers: CL = flag number to set DS = SYSDAT (BIOS data segment) Exit Registers: AX, BX, CX, DX are altered; all other registers preserved
INT_CHARSCAN	Call Far to this routine to have the BDOS check for the live control keys. Entry Registers: AL = character received from device DS = SYSDAT (BIOS data segment) Exit Registers: AL = character for BDOS to scan BL = 0 then discard the character BL = 1 then place AL in the input buffer for this device All other registers preserved

Table 4-1. (continued)

Function Explanation

INT_DISPATCH

Jump Far to this routine to exit the interrupt service routines and to force rescheduling of the currently ready to run processes. The BDOS assumes the DS register of the interrupted process is the first word on the stack followed by the three words pushed by the interrupt.

Entry Registers: DS = SYSDAT (BIOS data segment)
All registers, except DS, as on entry to the interrupt service routine. The original value of DS is the first word on the stack.

Exit Registers: This function does not return.

POLLED DEVICE DRIVERS

A polled I/O driver can execute CPU loops when waiting for a hardware event. This is inefficient and precludes keyboard type-ahead, live keyboard, and background programs (processes). Another type of polling calls the ?DISPATCH routine in the BIOS Kernel when waiting for a hardware event. This latter method allows background programs to run. Either kind of polling does not allow live keyboard, or keyboard type-ahead (or more generally buffered character I/O). Interrupt-driven character I/O allows these features to be implemented and is more efficient than polling. For these reasons, you are encouraged to use interrupt-driven device drivers instead of polling device drivers in the CP/M-86 Plus BIOS. Polling device drivers can be helpful, however, during BIOS development and debugging.

Do not call the INT_DISPATCH routine in the BDOS to give up the CPU when polling; instead, use the Kernel ?DISPATCH routine.

Some hardware events provide no status information from an interrupt or through a port that can be polled. Usually a specific amount of time must be delayed, then the hardware is assumed to be ready. An example is a diskette motor, which once turned on, must reach operational speed before being used. For this type of event, the BIOS Kernel ?DELAY function can be used to give up the CPU for a specified number of ticks. The ?DELAY function invokes the P_DELAY system call in the BDOS.

Since the CLOCK Module must support a system tick before P_DELAY works, drivers should be initially written with CPU loops for these time-outs, then replaced with calls to ?DELAY as one of the last steps in BIOS implementation.

End of Section 4

Section 5

System and BIOS Initialization

This section describes system initialization, the BIOS INIT Module, and device initialization.

SYSTEM INITIALIZATION

The CPM-86 Plus loader, CPMLDR, loads CPM3.SYS into memory and initializes DS to SYSDAT, then executes a JMPF (Jump Far instruction) to offset 0 in the BDOS code segment. This is the beginning of the BDOS initialization routine, which after performing internal system initialization, makes a CALLF (Call Far instruction) to offset 0 in the BIOS code segment. At offset 0 in the BIOS Code Header, a JMP BIOSINIT instruction starts the BIOSINIT routine. Section 11 discusses the CPMLDR more fully.

The BIOSINIT routine in the Kernel first calls ?INIT in the INIT module to perform any general hardware initialization needed. Then, the BIOSINIT routine calls the initialization routine specified in each DPH and CDB in the system. On entry to the Kernel BIOSINIT routine, the BDOS reserves 20 words on the stack for BIOS initialization; switch to a local stack if more space is needed by your initialization routines.

When the CDB and DPH initialization routines have been performed, BIOSINIT locates the character device that is the initial logical CONIN: device. This device is represented by the CDB whose offset is in the Kernel Data Header @BH_CIROOT variable. If a device is interrupt-driven, its associated CDB CDB_IINPUT field is equal to 0FFh and to 0 if not. The BIOSINIT routine copies the CDB_IINPUT field from the CONIN: CDB to the BIOS Data Header @BH_INTCONIN variable. The value of @BH_INTCONIN signals the BDOS whether the CONIN: device is interrupt-driven and is thus set to 0FFh if console input is interrupt-driven and to 0 if not.

BIOSINIT then calls ?CLOCK_INIT in the CLOCK module, and lastly prints the sign-on message using the Kernel public ?PMSG routine. The BIOSINIT routine next executes a RETF (Return Far instruction) back to the BDOS.

Some hardware initialization is often necessary in the bootstrap loader and CPMLDR. You may not have to duplicate this initialization in the BIOS.

BIOS INIT MODULE

The INIT Module contains the public ?INIT routine and defines the @SIGNON message. As described earlier, the Kernel BIOSINIT routine calls ?INIT during system initialization to perform any general

hardware initialization that is not accomplished by the subsequent CDB and DPH initialization routines or the clock initialization routine. A RET (Near Return instruction) must terminate the ?INIT routine.

The interrupt vectors in the first Kbyte of memory are initialized by the following sequence. The BDOS sets interrupt vector 224 to point to the normal BDOS entry before calling BIOSINIT. After BIOSINIT returns, the BDOS reinitializes interrupt 224 and copies interrupt vectors 0, 1, 3, 4, 224, and 225 to a local save area.

The BDOS copies the saved interrupt vectors 0, 1, 3, 4, 224, and 225 into the interrupt vectors in low memory during each P_CHAIN or P_TERM system call. Thus, whenever a program chains or terminates, these six interrupt vectors are reinitialized.

The BDOS keeps copies of interrupt vectors 0, 1, 3, 4, 224, and 225 for each process and reinitializes the interrupt vectors in low memory before a process is given the CPU.

The ?INIT routine usually initializes all interrupt vectors to point to an interrupt trap routine that prevents spurious interrupts from vectoring to an unknown location. The interrupt trap routine usually prints out an error message, enables interrupts, and performs a HLT (Halt instruction). The CPU is halted since the integrity of the operating system image is not guaranteed after an uninitialized interrupt.

The device CDB_INIT and DPH_INIT routines for each CDB and DPH device as well as the CLOCK_INIT routine then set the specific interrupt vectors they need. All interrupt vectors should be initialized when BIOSINIT returns to the BDOS. However, during debugging several interrupt vectors are left unchanged to allow CP/M-86 1.X and DDT-86 to monitor your CP/M-86 Plus BIOS. Section 10 examines debugging.

The BIOS INIT module can set the 8087 variable in the BIOS Kernel Data Header. If the 8087 exists set the @BH_8087 byte to 0FFh.

DEVICE INITIALIZATION

The Kernel BIOSINIT routine performs character and disk device initialization by calling the INIT routines indicated in all the DPHs and CDBs. BIOSINIT makes no initialization call for DPHs and CDBs whose fields in the BIOS Kernel Data Header are zero; these devices are considered unsupported by the BIOS.

If several DPHs or CDBs share the same physical device, the DPHs or CDBs cooperate so as not to reinitialize the same device or allocate extra flags for interrupt operations. For instance, a floppy disk controller that can perform I/O operations to several drives can be shared by several DPHs.

If a driver is interrupt-driven and therefore requires one or more system flags, the specific device INIT routine allocates a system flag for itself. This is done by accessing the @BH_NEXTFLAG and @BH_LASTFLAG fields in the BIOS Data Header. During BIOS initialization, the next unused flag number is present in the @BH_NEXTFLAG field. The driver must save this flag number and use it when performing wait and set flag operations. The driver initialization routine must also increment the @BH_NEXTFLAG field to reserve the flag, and thus indicate the next available flag number. @BH_LASTFLAG contains a value that indicates the last available system flag number. If @BH_LASTFLAG is less than @BH_NEXTFLAG, no more flags are available. The initialization routine for an interrupt-driven device must ensure a flag is indeed available. GENCPM sets @BH_NEXTFLAG and @BH_LASTFLAG at system generation time.

End of Section 5

Section 6

Character I/O

This section describes the CP/M-86 Plus BIOS Character I/O routines you supply for a specific machine. The first subsection describes the Character Device Block, a data structure you use to define character devices in the BIOS. The next subsection describes the hardware specific routines associated with a device and its Character Device Block. Another subsection presents character I/O buffering, including type-ahead and live keyboard. The final subsection covers character I/O error messages.

All character I/O drivers supporting different kinds of devices, such as serial and parallel printers and serial and memory-mapped CRTs, can be contained in one module or broken up as convenient into several modules. The example BIOS supports all of the character I/O devices in the one module, CHARIO.A86. The CHARIO.A86 file is a useful reference while reading this section.

As for all of the BIOS modules, the Character I/O modules must consist of separate code and data. Each character I/O device must have a Character Device Block (CDB).

The BDOS passes eight-bit data to and from the character IO_ functions in the Kernel. If necessary, the character device driver must mask the most significant (parity) bit.

CHARACTER DEVICE BLOCK (CDB)

Each character I/O device has an associated Character Device Block (CDB) that contains information about the character device. Throughout this manual and the example BIOS, fields in the CDB are prefixed with "CDB_". The following listing shows the CDB format and is also included in the file CDB.LIB on your distribution diskette.

Listing 6-1. Character Device Block Format

```
;*******************************************************
;
;       Console Device Block Equates
;
;*******************************************************
;
;       +------+------+------+------+------+------+------+------+
; 00h:  |                    NAME                    |   SUPCHAR   |
;       +------+------+------+------+------+------+------+------+
; 08h:  |   CURCHAR   |SUPOEM|CUROEM| TXB  | RXB  | TYPE |IINPUT|
;       +------+------+------+------+------+------+------+------+
; 10h:  | FLAGS| RESRV|    COLINK   |    AOLINK   |    LOLINK   |
;       +------+------+-------------+-------------+-------------+
; 18h:  |    INIT     |    INPUT    |   INSTAT    |   OUTPUT    |
;       +------+------+------+------+------+------+------+------+
; 20h:  |   OUTSTAT   |
;       +------+------+

CDB_NAME        equ     byte ptr 0
CDB_SUPCHAR     equ     word ptr 6
CDB_CURCHAR     equ     word ptr 8
CDB_SUPOEM      equ     byte ptr 10
CDB_CUROEM      equ     byte ptr 11
CDB_TXB         equ     byte ptr 12
CDB_RXB         equ     byte ptr 13
CDB_TYPE        equ     byte ptr 14
CDB_IINPUT      equ     byte ptr 15
CDB_FLAGS       equ     byte ptr 16
CDB_RESRV       equ     byte ptr 17
CDB_COLINK      equ     word ptr 18
CDB_AOLINK      equ     word ptr 20
CDB_LOLINK      equ     word ptr 22
CDB_INIT        equ     word ptr 24
CDB_INPUT       equ     word ptr 26
CDB_INSTAT      equ     word ptr 28
CDB_OUTPUT      equ     word ptr 30
CDB_OUTSTAT     equ     word ptr 32
```

Listing 6-2 shows an example CDB definition from the CHARIO.A86 file. (The CRT0_CS, CRT0_CC, CRT0_CT values are equates defined in CHARIO.A86. The symbols crt0_init, crt0_input, crt0_instat, crt0_output, crt0_outstat are routines in CHARIO.A86. The BAUD_9600 symbol is defined in the CDB.LIB file.)

Listing 6-2. Example CDB Definition

```
@cdba   db      'CRT0   '       ;name
        dw      CRT0_CS         ;supported characteristics
        dw      CRT0_CC         ;current characteristics
        db      0,0             ;no OEM characteristics
        db      BAUD_9600       ;transmit baud
        db      BAUD_9600       ;receive baud
        db      CRT0_CT         ;type of device
        db      0FFh            ;will support type ahead
        db      2               ;2 flags used
        db      0               ;reserved
        dw      0               ;console output link
        dw      0               ;aux output link
        dw      0               ;list output link
        dw      crt0_init       ;device A init
        dw      crt0_input      ;device A input
        dw      crt0_instat     ;device A input status
        dw      crt0_output     ;device A output
        dw      crt0_outstat    ;device A output status
```

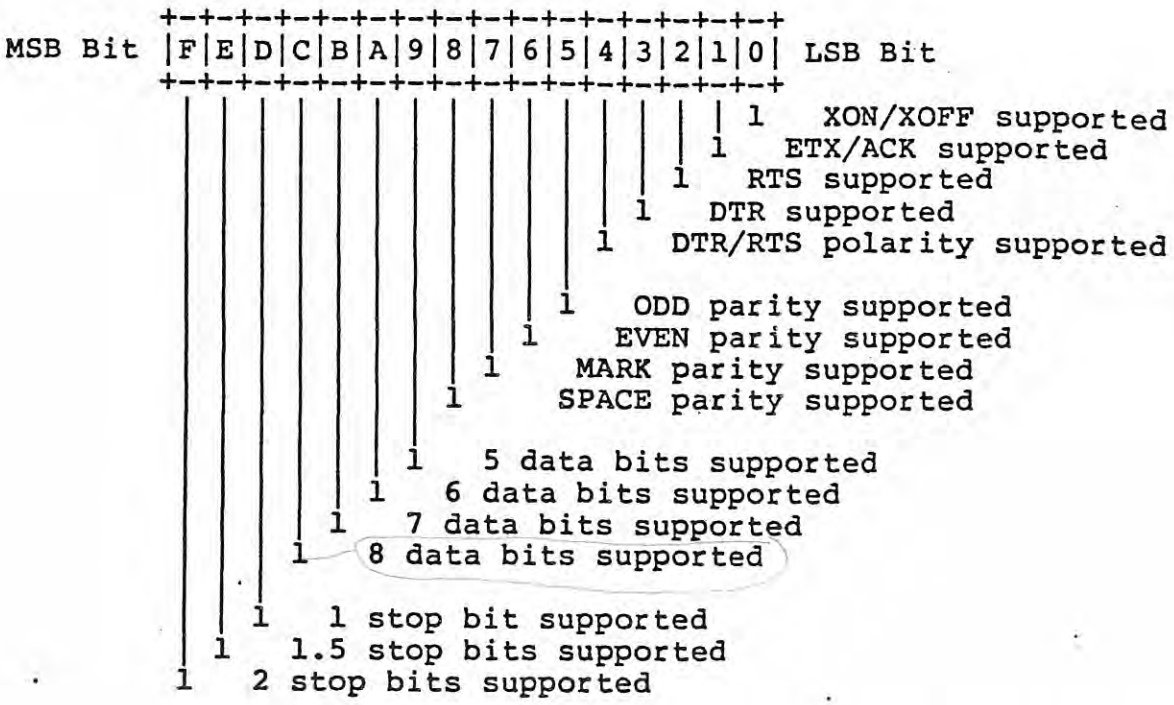
Table 6-1 examines each CDB field:

Table 6-1. Character Device Block Data Fields

Data Field	Explanation
CDB_NAME	[Six-character name of this physical device] The device name must be in capital alphanumeric ASCII characters, left-justified in the field, and padded on the right with ASCII spaces.
CDB_SUPCHAR	[Supported characteristics] This field indicates the device characteristics supported by the driver associated with the CDB. The possible set of device characteristics are start bits, parity, line polarity, protocols, and data bits. Supported characteristics are indicated by setting the appropriate bits in the CDB_SUPCHAR field. These bits are assigned as follows:

Table 6-1. (continued)

Data Field Explanation

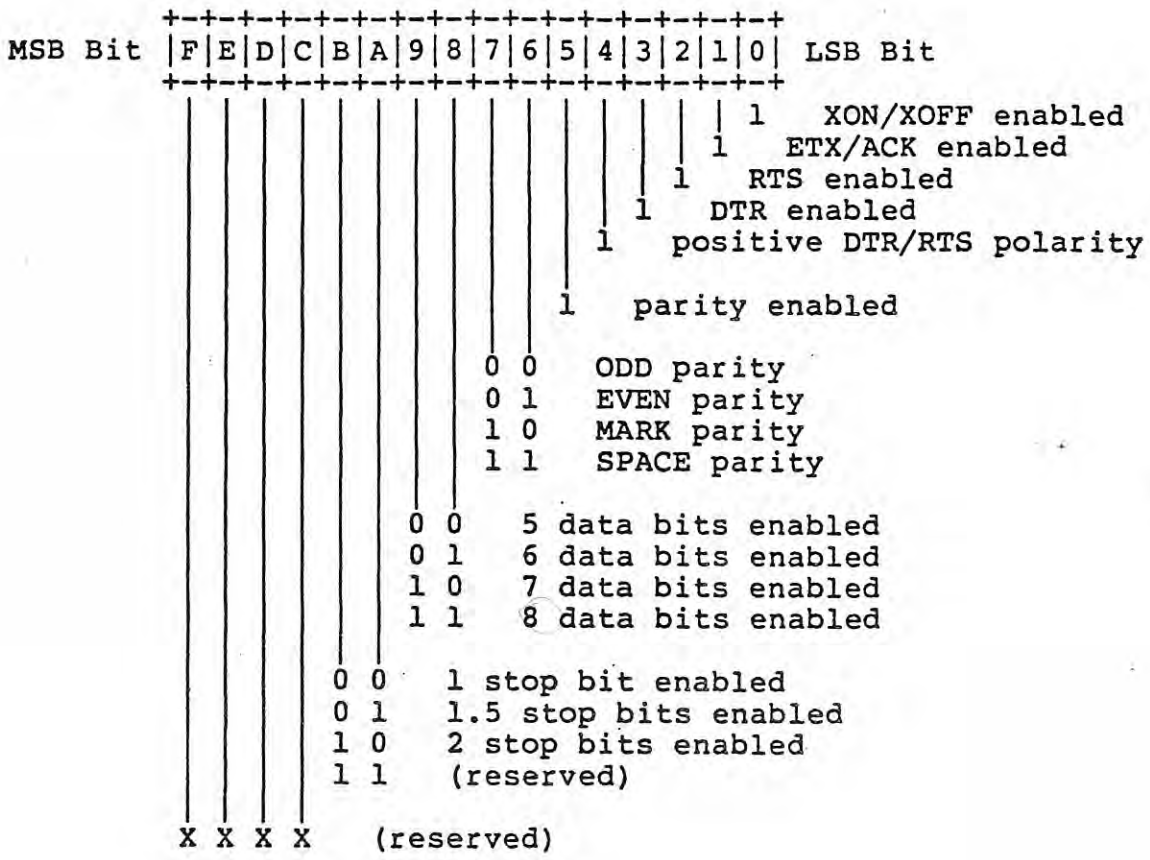


Note the DTR/RTS polarity field signifies that polarity is supported but not whether it is positive or negative. Equates for these bits in the CDB_SUPCHAR field are found in the CDB.LIB file.

CDB_CURCHAR [Current characteristics] This field specifies the characteristics the device driver is currently using. Note this field does not correspond one-to-one with the bits in CDB_SUPCHAR. The parity, data bits, and stop bits are condensed into binary values in CDB_CURCHAR. The CDB_INIT routine can thus mask, shift, and jump based on these CDB_CURCHAR bits. Equates for these operations on the CDB_CURCHAR field are in the CDB.LIB file.

Table 6-1. (continued)

Data Field Explanation



Note the DTR/RTS polarity (bit 4) has meaning only when the corresponding bit is set in the CDB_SUPCHAR field. When bit 4 of the CDB_SUPCHAR field is set, the DTR/RTS polarity is negative when bit 4 of the CDB_CURCHAR field is 0 and positive when bit 4 of the CDB_CURCHAR field is 1.

The DEVICE utility alters this field to change the current device characteristics, then calls the CDB_INIT routine.

CDB_SUPOEM [Supported OEM characteristics] This field is defined by the OEM for any device characteristics and protocols that can not be represented with the CDB_SUPCHAR field. Set CDB_SUPOEM and CDB_CUROEM to 0 if there are no OEM-defined characteristics. The DEVICE utility displays this field, but is otherwise unused by CP/M-86 Plus.

Table 6-1. (continued)

Data Field	Explanation
CDB_CUROEM	[Current OEM characteristics] This field contains the OEM-defined characteristics the device driver associated with the CDB is currently using. This field is defined by the OEM when CDB_SUPOEM is defined. The DEVICE utility alters this field to change the current OEM device characteristics, then calls the CDB_INIT routine. DEVICE assumes the definitions of bits in CDB_SUPOEM are one to one with the bits in CDB_CUROEM.
CDB_TXB	[Transmit baud] This is the value representing the current transmit baud of the device associated with the CDB.
CDB_RXB	[Receive baud] This is the value representing the current receive baud of the device associated with the CDB. If the CDB_TYPE field (below) has the CI_SOFTBAUD bit set, the DEVICE utility can change the baud by setting the CDB_TXB and CDB_RXB fields, and then calling the CDB_INIT routine. See the discussion on the CDB_INIT routine in the "Character Device Block Routines" subsection below, regarding unsupported baud settings. The following are the values that CDB_TXB and CDB_RXB can assume (also in the CDB.LIB file as equates):

SYMBOL	VALUE	EXPLANATION
BAUD_NONE	00h	No baud rate for this device
BAUD_50	01h	50 baud
BAUD_625	02h	62.5 baud
BAUD_75	03h	75 baud
BAUD_110	04h	110 baud
BAUD_1345	05h	134.5 baud
BAUD_150	06h	150 baud
BAUD_200	07h	200 baud
BAUD_300	08h	300 baud
BAUD_600	09h	600 baud
BAUD_1200	0Ah	1200 baud
BAUD_1800	0Bh	1800 baud
BAUD_2000	0Ch	2000 baud
BAUD_2400	0Dh	2400 baud
BAUD_3600	0Eh	3600 baud
BAUD_4800	0Fh	4800 baud

BAUD_7200	10h	7200 baud
BAUD_9600	11h	9600 baud
BAUD_192	12h	19200 baud
BAUD_384	13h	38400 baud
BAUD_56	14h	56000 baud
BAUD_768	15h	76800 baud
BAUD_OEM1	16h	OEM-defined
BAUD_OEM2	17h	OEM-defined
BAUD_OEM3	18h	OEM-defined

Table 6-1. (continued)

Data Field	Explanation
CDB_TYPE	[Device type] The CDB_TYPE byte specifies whether the device is an input or output device, whether it has a selectable baud rate, and whether it is a serial device. The following bits are defined for this field and are also included in the CDB.LIB file.

SYMBOL	VALUE	EXPLANATION
CT_INPUT	01H	Device performs input
CT_OUTPUT	02H	Device performs output
CT_SOFTBAUD	04H	Software-selectable baud rate
CT_SERIAL	08H	Serial device

CDB_IINPUT	[Interrupt input] Set this field to 0FFH if the device associated with the CDB is interrupt-driven on input; otherwise, set the field to 0. The BIOSINIT routine and the DEVICE utility use this field to set the @BH_CONINT in the BIOS Kernel Data Header. The @BH_CONINT field is set to the CDB_IINPUT value of the CDB on the console input root, indicating to the BDOS whether console input is interrupt-driven.
CDB_FLAGS	This field is initialized to the maximum number of flags this device needs for ?WAITFLAG and INT_SETFLAG operations. Usually there is a flag per interrupt service routine used by a driver. For example, if your console input is interrupt-driven, but console input status, output, and output status are not, then you need one flag for the console I/O driver. GENCPM uses this field in calculating the minimum number of flags needed in the system.
CDB_RESRV	This field is reserved for future use.

Table 6-1. (continued)

Data Field	Explanation
CDB_COLINK	This is the offset of the next CDB representing the next device attached to the logical device CONOUT: via the list beginning at @BH_COROOT. The CDB_COLINK field of the last CDB in the list, is set to 0.
CDB_AOLINK	This is the offset of the next CDB representing the next device attached to the logical device AUXOUT: via the list beginning at @BH_AOROOT. The CDB_AOLINK field of the last CDB in the list, is set to 0.
CDB_LOLINK	This is the offset of the next CDB representing the next device attached to logical device LST: via the list beginning at @BH_LOROOT. The CDB_LOLINK field of the last CDB in the list, is set to 0.
CDB_INIT	This is the offset of the initialization routine for this device. The initialization routine is responsible for setting the protocol and baud rate, if applicable, for this device and performing any other necessary initialization required. The first time CDB_INIT is called register DL is set to 0 by the BIOS Kernel. The CDB_INIT routine must program the device to correspond to the specifications in CDB_CURCHAR, CDB_RXB, CDB_TXB, and CDB_CUROEM fields. CDB_INIT returns AX = 0 if there is no error is setting the device to these specifications, and AX = 0FFFFh if there is an error. On entry, DS:BX specifies the address of the CDB for this device. "Character Device Block Routines" in this section supplies more complete information on this and the following CDB routines.
CDB_INPUT	This is the offset of the character-input routine for this device. On entry, DS:BX specifies the address of the CDB for this device.
CDB_INSTAT	This is the offset address of the character-input status routine for this device. On entry, DS:BX specifies the address of the CDB for this device.

CDB_OUTPUT	This is the offset address of the character-output routine for this device. On entry, DS:BX specifies the address of the CDB for this device.

Table 6-1. (continued)

Data Field	Explanation
CDB_OUTSTAT	This is the offset address of the character-output status routine for this device. If character protocols are supported, they must not hang the device. Status routines must always return immediately. On entry, DS:BX specifies the address of the CDB for this device.

CHARACTER DEVICE BLOCK (CDB) ROUTINES

The Character Device Block fields CDB_INIT, CDB_INPUT, CDB_INSTAT, CDB_OUTPUT, and CDB_OUTSTAT are defined by the system implementor to be the offsets the routines that perform the functions indicated by the field names. Section 3 explains how the BIOS Kernel calls these CDB routines. These five CDB routines along with the CDB itself constitute a character I/O driver for one device. Generally, the CDB routines are not shared among different drivers since they are usually specific to the physical device. Each of these fields must be initialized with the offset of a valid routine, even if the routine only performs a RET (Return instruction).

The CDB routines follow must follow certain conventions when a devices is input only or output only. For example, there is usually no input associated with a list device. In this case, when a device is output only the device's CDB_INPUT routine is defined to return a null (0) and the its CDB_INSTAT routine is defined to return a false status (AL=0). When a device is input only, the CDB_OUTPUT routine simply returns, and the CDB_OUTSTAT routine is defined to return a true status (AL=0FFh).

CDB_INIT Routine

The CDB_INIT routine initializes the I/O hardware for the device. The BIOS Kernel BIOSINIT routine calls the CDB_INIT routine for each CDB in the BIOS Kernel Data Header. The DEVICE utility calls the Kernel IO_DEVINIT function, which also calls CDB_INIT for a specific CDB. The DEVICE utility does this after changing the protocol or baud of the device, in other words, after changing any of the CDB fields CDB_CURCHAR, CDB_CUROEM, CDB_TXB, or CDB_RXB.

If CDB_INIT is called and the device hardware cannot be set in accordance with the latter CDB fields, CDB_INIT should return an error (AX=0FFFFh). An error can occur if the baud selected is unsupported or if there is a hardware problem. When the CDB_TXB or CDB_RXB fields are set to values representing unsupported bauds, CDB_INIT should leave the hardware baud setting unaltered and return an error. The DEVICE utility recognizes the error return and restores the original values of CDB_TXB and CDB_RXB.

The Kernel BIOSINIT routine sets register DL to 1 before calling CDB_INIT, and the Kernel IO_DEVINIT function sets DL to 0 before calling CDB_INIT. This allows the CDB_INIT routine to distinguish the first initialization call from subsequent ones. Any one-time initialization code, such as allocating flags (using @BH_NEXTFLAG and @BH_LASTFLAG), should be skipped on all CDB_INIT calls but the first.

The register conventions between the BIOS Kernel and the CDB_INIT routines are as follows:

```
Entry Registers:  DL = 0 if first time initialization
                  DL = 1 if not first time initialization
                  BX = offset of CDB
                  DS = SYSDAT (BIOS data segment)
                  ES = process environment

Exit Registers:   AX = 0 if no error
                  AX = 0FFFFh if error
                  DS, ES preserved
```

CDB_INPUT Routine

The CDB_INPUT routine for each character device reads a character from the device or the input buffer at the request of the BIOS Kernel. Type-ahead requires the use of an input buffer (see "Character Input Interrupt" later in this section).

CDB_INPUT should return a null (0) when the device is output only. The most significant (parity) bit of the character is preserved by the Kernel and the BDOS, and if parity from this device is not desired, the CDB_INPUT routine must mask it off. The register conventions between the BIOS Kernel and the CDB_INPUT routines are as follows:

```
Entry Registers:  BX = offset of CDB
                  DS = SYSDAT (BIOS data segment)
                  ES = process environment

Exit Registers:   AL = character
                  DS, ES preserved
```

CDB_INSTAT Routine

The CDB_INSTAT routine for each character device returns the device input status at the request of the BIOS Kernel. The CDB_INSTAT returns a true (AL=0FFh) value if a character is ready from the device, or if any characters are available from the device's input buffer.

CDB_INSTAT should return a false status (AL=0) when the device is output only. The register conventions between the BIOS Kernel and the CDB_INSTAT routines are the following:

```
Entry Registers:  BX = offset of CDB
                  DS = SYSDAT (BIOS data segment)
                  ES = process environment
```

Exit Registers: AL = 0FFh if character ready
AL = 0 if character not ready
DS, ES preserved

CDB_CHAROUT Routine

The CHAROUT routine for each character device sends a character to the associated device at the request of the BIOS Kernel.

CDB_OUTPUT should simply return when the device is input only. The most significant (parity) bit of the character is preserved by the BIOS Kernel and the BDOS; if parity cannot be sent to this device the CDB_OUTPUT routine must mask it off. The register conventions between the BIOS Kernel and the CDB_OUTPUT routines are the following:

```
Entry Registers:  CL = character to send to device
                  BX = offset of CDB
                  DS = SYSDAT (BIOS data segment)
                  ES = process environment

Exit Registers:   DS, ES preserved
```

CDB_OUTSTAT Routine

The CDB_OUTSTAT routine for each character device returns the output status of the associated device at the request of the BIOS Kernel. When output is interrupt-driven, the output status is true (AL=0FFh) if there is space in output buffer. When the device is not ready, or in the interrupt-driven case when there is no room in the output buffer, CDB_OUTSTAT returns a false status (AL=0). The next subsection covers interrupt-driven character devices in more detail.

CDB_OUTSTAT returns a true status (AL=0FFh) when the device is input only. The register conventions between the BIOS Kernel and the CDB_OUTSTAT routines are as follows:

```
Entry Registers:  BX = offset of CDB
                  DS = SYSDAT (BIOS data segment)
                  ES = process environment

Exit Registers:   AL = 0FFh if device is ready
                  AL = 0 if not ready
                  DS, ES preserved
```

INTERRUPT-DRIVEN CHARACTER I/O

Either or both character input and character output can be interrupt-driven. As discussed in Section 4, interrupt drivers are more efficient and allow several features to be present in CP/M-86 Plus that are not possible with polling. Read Section 4 before reading this material.

Each interrupt-driven character I/O device typically makes use of two character buffers, one for input and one for output. The device

input interrupt service routine fills the input buffer and processes calling the CDB_INPUT routine empty it. Processes calling the CDB_OUTPUT routine fill the output buffer, and the device output interrupt service routine empties the output buffer.

The process and the interrupt stop and start each other when a character or buffer space is not available using ?WAITFLAG and INT_SETFLAG operations.

Each character interrupt service routine usually keeps a local variable indicating if a flag set operation is necessary. This provides even more efficient I/O and is discussed later in this section.

Character Input Interrupt (type-ahead)

A console input interrupt service routine in conjuction with the CDB_INSTAT and CDB_INPUT routines for a particular device, can implement type-ahead and live keyboard. Listing 6-3, below provides an example implementation of a CDB_INSTAT routine, a CDB_INPUT routine, and a character input interrupt service routine. These routines support type-ahead and live keyboard and are part of the CHARIO.A86 file on the distribution disk.

A device using interrupt-driven input must have the CDB_IINPUT field set to 0FFh in its CDB. When this CDB is attached to the CONIN: logical device by putting the offset of the CDB in @BH_CIROOT, the CDB_IINPUT value is copied to the @BH_CONINT field in the BIOS Kernel Data Header. The value in @BH_CONINT informs the BDOS that console input is interrupt-driven. When @BH_CONINT is 0FFh, the BDOS does not make BIOS IO_CONST calls to check for CTRL-C, CTRL-S, CTRL-Q, and CTRL-P.

When the device is attached to CONIN:, the device input interrupt service routine must perform a CALLF (Call Far instruction) to the INT_CHARSCAN functions in the BDOS with every character received from the input device. (The address of INT_CHARSCAN is in SYSDAT, shown in Appendix C.) An interrupt service routine can determine if the CDB that represents the interrupting device is attached to CONIN: by comparing the offset of the CDB with @BH_CIROOT. The register conventions for the INT_CHARSCAN function are shown in Section 4.

The input interrupt handler in Listing 6-3 CRT0_INPUT_INT, checks for a CTRL-C when the INT_CHARSCAN function returns register BL equal to 0, signifying the character should be discarded. When this occurs the CTRL-C function was not disabled by the C_MODE or C_RAWIO system calls (See the Programmer's Guide) and the BDOS terminated the running program. When this occurs it is usually desirable to flush the type-ahead buffer as shown in CRT0_INPUT_INT.

At system initialization, the BIOSINIT routine ensures that @BH_INTCONIN is set in accordance with the CDB addressed by @BH_CIROOT. Generally, the DEVICE utility is the only way the

logical assignments can be subsequently changed. When DEVICE removes or replaces CDBs from @BH_CIROOT, it updates the @BH_INTCONIN field in the BIOS Kernel Data Header.

The input buffer shared by the CDB_INPUT routine and the interrupt service routine must be protected from simultaneous access. In Listing 6-3, interrupts are disabled in the CRT0_INPUT routine when a process tests for and removes characters in the buffer. (The CRT0_INPUT routine is in the CDB_INPUT routine for CDBA.) Interrupts are enabled in the interrupt service routine since the @BH_ININT (in interrupt count) guarantees a process cannot execute until the interrupt service routine is complete. If you do not reenable interrupts in any of interrupt service routines within the BIOS, the @BH_ININT byte does not need to be used.

The CDB_INPUT and the interrupt service routines should keep a local variable to indicate whether the interrupt routine needs to perform a CALLF (Call Far instruction) to INT_SETFLAG in the BDOS. When the CDB_INPUT routine finds no input characters in the buffer, it sets the local variable, then performs a CALL (Call Near instruction) to ?WAITFLAG in the BIOS Kernel. The interrupt service routine checks this variable, and performs a CALLF to INT_SETFLAG if a ?WAITFLAG is being or has been executed. Thus a CALLF INT_SETFLAG is executed only when necessary. (The register conventions for the BDOS INT_SETFLAG are shown in Section 4.)

This "flag waiting" variable is usually kept in the character buffer structure along with the character count and pointers into buffer. The CDB_INPUT routine must check for characters in the buffer and set the "flag waiting" variable with interrupts off.

In Listing 6-3, when the interrupt routine finds there is no more room in the input buffer (the type-ahead buffer is full), characters are not saved and are lost. If desired the user can be notified via a light on the keyboard, a message on a screen status line, or by a bell tone.

Listing 6-3 assumes the @CDBA definition from Listing 6-2. An input buffer structure is also defined in this example. Equates for this structure begin with the letters "BUF_". The BUF_FLAGNO field is the system flag used for ?WAITFLAG and INT_SETFLAG operations and is assumed to have been previously allocated and set by the first call to the CDBA_INIT routine for this device.

Listing 6-3. Buffered Interrupt-driven Character Input

```
; The following equates define a buffer descriptor used to manage
; circular input and output buffers.  The buffer size must be a power
; of 2 since the next buffer position is calculated with an AND instructic

BUF_LEN         equ     256             ;use immediate value
                                        ;for efficiency
```

```
BUF_FLAGN       equ     byte ptr 0      ;system flag number to use
BUF_FWAIT       equ     byte ptr 1      ;0FFh if process is flag wait
BUF_COUNT       equ     word ptr 2      ;chars in buffer
BUF_CHAROUT     equ     word ptr 4      ;number of next char to take out
BUF_BUFFER      equ     byte ptr 6      ;first byte of buffer

CSEG

crt0_instat:    ;Input status routine for the CRT0 device
;===========
;       Entry:  BX = CDB address
;       Exit:   AL = 0FFh if character ready
;                  = 0 if no character ready
;               BX = input buffer offset

        mov bx,offset in_buf_desc       ;set BX to input buffer for
        xor ax,ax                       ;this device
        cmp ax,BUF_COUNT[bx]            ;is buffer is empty ?
        je cis_empty
          dec ax
cis_empty:
        ret

crt0_input:     ;character input routine for the CRT0 device
;==========
;       Entry:  BX = CDB address
;       Exit:   AL = character
;               BX = input buffer offset

        cli                             ;disable CPU interrupts
        call crt0_instat                ;is there a char in the buffer?
        test al,al ! jnz ci_ready       ;if not then wait on flag
          mov BUF_FWAIT[bx],0FFh        ;request CALLF INT_SETFLAG
          sti                           ;from interrupt
          mov dl,BUF_FLAGN[BX]          ;flag number for this input devic
          call ?waitflag
          jmps crt0_input               ;test status again to be sure
ci_ready:                               ;get char out of buffer
        mov si,BUF_CHAROUT[bx]          ;offset in buffer of next out cha
        mov al,BUF_BUFFER[bx+si]        ;get the next character
        inc si ! and si,BUF_LEN-1       ;back to 0 if past end of buffer
        mov BUF_CHAROUT[bx],si          ;update next number of next char
        dec BUF_COUNT[bx]               ;one less char in buffer
        sti                             ;enable CPU interrupts
        ret

crt0_input_int:         ;Input interrupt service routine for the CRT0 dev
;==============
;       Entry:  IP,CS,CPU FLAGS on stack, interrupts off
;       Exit:   all registers preserved

        push ds ! mov ds,@sysdat        ;save DS on process stack
        inc @bh_inint                   ;stop dispatches
        mov crt0_in_ss,ss               ;switch stacks
```

```
        mov crt0_in_sp,sp
        mov ss,@sysdat                  ;DS and SS = BIOS data segment
        mov sp,offset crt0_in_tos
        sti                             ;enable interrupts
        push ax ! push bx               ;save registers that will be alte
        in al,SS_STATUS                 ;check status again to ensure
        test al,SS_RECV_READY           ;character ready
        jz cii_done
          in al,SS_DATA                 ;get the character
          cmp @bh_ciroot,offset @cdba
          jne cii_save_char             ;is this the CONIN: device ?
            callf INT_CHARSCAN          ;yes - let the BDOS check the cha
            test bl,bl                  ;if BL=0 don't save the char
            jnz cii_save_char
              cmp al,CTRL_C             ;if char is a control-C
              jne cii_done              ;discard type ahead buffer
                mov in_buf_desc+BUF_COUNT,0
                jmps cii_done
cii_save_char:
          mov bx,offset in_buf_desc     ;put char in buffer if not full
          push cx ! mov cx,BUF_COUNT[bx]
          cmp cx,BUF_LEN ! jae cii_full ;if buffers full ignore char
            push si
            mov si,BUF_CHAROUT[bx]      ;find next free byte
            add si,cx                   ;in the buffer
            and si,BUF_LEN-1            ;back to 0 if past end of buffer
            mov BUF_BUFFER[si+bx],al    ;store char
            inc cx                      ;bump char counter
            mov BUF_COUNT[bx],cx
            pop si
cii_full:
          cmp BUF_FWAIT[bx],0FFh        ;is a process waiting on flag ?
          jne cii_donel
            mov BUF_FWAIT[bx],0         ;yes - set the flag
            mov dl,BUF_FLAGN[bx]        ;DL=flag for this input device
            push dx                     ;?SETFLAG alters AX,BX,CX,DX
            callf int_setflag           ;AX and BX already saved
            pop dx
cii_donel:
        pop cx
cii_done:
      pop bx
      cli                               ;reset the PIC's
      mov al,NS_EOI
      out MASTER_PIC_PORT,al            ;PIC ports for Compupro
      out SLAVE_PIC_PORT,al
      pop ax
      mov ss,crt0_in_ss
      mov sp,crt0_in_sp
      dec @bh_inint                     ;if in interrupt count
      jnz cii_exit                      ;is 0 then dispatch
        jmpf int_dispatch
cii_exit:
      pop ds                            ;otherwise return to the
      iret                              ;previous interrupt service routi
```

```
DSEG

        extrn   @bh_inint:byte          ;in interrupt count in BIOS
                                        ;Kernel Data Header

;       console input interupt stack area

crt0_in_sp      rw      1
crt0_in_ss      rw      1
                rw      32
crt0_in_tos     rw      0

in_buf_desc     rb      1               ;flag number - set by CDB_INIT
                db      0               ;"flag waiting" variable
                dw      0               ;number of chars in buffer
                dw      0               ;next char to take out of buffer
                rb      BUF_LEN         ;buffer
```

Interrupt Character Output

Interrupt character output consists of a CDB_OUTPUT routine putting characters into a buffer and an interrupt service routine taking them out and sending them to the device. Listing 6-4 below, shows an example implementation of buffered interrupt-driven character output. It is also found in the CHARIO.A86 file on the distribution disk.

In Listing 6-4, when the CRT0_OUTPUT routine is first executed, a character is sent directly to the device. (CRT0_OUTPUT is the CDB_OUTPUT routine for CDBA.) During the time the device is sending the character, processes calling CRT0_OUTPUT fill the output buffer for the device. The device generates an interrupt when it is ready to send another character. The interrupt service routine takes a character out of the output buffer and sends it to the device. The last interrupt generated from the device finds nothing in the buffer and output stops. The next character sent to CRT0_OUTPUT goes directly to the device, starting the sequence over again.

The CRT0_OUTPUT routine calls ?WAITFLAG when there is no room in the buffer. The interrupt service routine executes a CALLF INT_SETFLAG to the BDOS when a process is "flag waiting" and there is at least one space for a character in the buffer. Characters cannot be lost on output; thus, the program generating output characters must wait until buffer space is available.

Using a local variable to record whether a process is waiting on a flag (or on the way to waiting) makes console output more efficient. Note interrupts on the CPU are disabled in the CRT0_OUTPUT routine when testing the state of the buffer, and the device ready status before deciding if the character goes to the device or into the buffer. Interrupts are enabled in the interrupt service routine CRT0_OUTPUT_INT, shown in Listing 6-4 since the @BH_ININT (in

interrupt count) guarantees a process cannot execute until the interrupt service routine is complete. If you do not reenable interrupts in any of interrupt service routines within the BIOS, the @BH_ININT byte does not need to be used.

The interrupt service routine can further be "tuned" for performance by changing the buffer size and by not making the CALLF to INT_SETFLAG until more of the output buffer is empty. The CRT0_OUTPUT_INT interrupt service routine waits for half of the buffer to empty before performing the CALLF to INT_SETFLAG.

The CDB defined in Listing 6-2 is assumed for the code in Listing 6-4. An output buffer structure is also defined in this example. Equates for this structure begin with the letters "BUF_". The BUF_FLAGNO field is the system flag used for ?WAITFLAG and INT_SETFLAG operations and is assumed to have been previously allocated and set by the first call to the CDBA_INIT routine for this device.

Listing 6-4. Buffered Interrupt-driven Character Output

```
; The following equates define a buffer descriptor used to manage
; circular input and output buffers.  The buffer size must be a power
; of 2 since the next buffer position is calculated with an AND instruct

BUF_LEN         equ     256             ;use immediate value
                                        ;for efficiency

BUF_FLAGN       equ     byte ptr 0      ;system flag number to use
BUF_FWAIT       equ     byte ptr 1      ;0FFh if process is flag waiting
BUF_COUNT       equ     word ptr 2      ;chars in buffer
BUF_CHAROUT     equ     word ptr 4      ;number of next char to take out
BUF_BUFFER      equ     byte ptr 6      ;first byte of buffer

CSEG

crt0_outstat:   ;character output routine for CRT0 device
;============
;       Entry:  BX = CDB address
;       Exit:   AL = 0FFh if character ready
;                  = 00h if character not ready
;               BX = offset of output buffer

        mov bx,offset out_buf_desc      ;get offset of output buffer
        xor ax,ax
        cmp BUF_COUNT[bx],BUF_LEN       ;compare char count with size of
        jae cos_full                    ;is buffer full ?
          dec ax                        ;no - return ready
cos_full:
        ret

crt0_output:    ;character output routine for CRT0 device
;===========
;       Entry:  CL = character
;               BX = CDB address
;       Exit:   None

        push cx                         ;save char to output
co_stat:
        cli
        call crt0_outstat               ;call output status
        test al,al ! jnz co_ready       ;check space in buffer
          mov BUF_FWAIT[bx],0FFh        ;request CALLF INT_SETFLAG
          sti
          mov dl,BUF_FLAGN[bx]          ;from interrupt
          call ?waitflag
          jmps co_stat                  ;check status again to be sure
co_ready:
        cmp BUF_COUNT[bx],0             ;is buffer empty and
        jne co_putchar
          in al,SS_STATUS               ;device ready ?
          test al,SS_TRANS_READY
          jz co_putchar
```

```
            pop ax                          ;AL = char to output
            out SS_DATA,al                  ;yes - send directly to device
            jmps co_ret
co_putchar:
        pop ax                              ;AL = char to output
        mov si,BUF_CHAROUT[bx]              ;put char in buffer
        add si,BUF_COUNT[bx]                ;next free buffer space
        and si,BUF_LEN-1                    ;back to 0 if past end of buffer
        mov BUF_BUFFER[si+bx],al            ;store char
        inc BUF_COUNT[bx]                   ;bump char counter
co_ret:
        sti                                 ;enable CPU interrups
        ret

crt0_output_int:        ;Output interrupt service routine for CRT0 device
;===============
;       Entry:  IP,CS,CPU flags on stack, interrupts off
;       Exit:   all registers preserved

        push ds ! mov ds,@sysdat            ;save DS on process stack
        inc @bh_inint
        mov crt0_out_ss,ss                  ;switch stacks
        mov crt0_out_sp,sp
        mov ss,@sysdat                      ;DS and SS = BIOS data segment
        mov sp,offset crt0_out_tos
        sti                                 ;enable interrupts
        push ax ! push bx                   ;save on local stack
        mov bx,offset out_buf_desc
        in al,SS_STATUS                     ;ensure hardware is ready
        test al,SS_TRANS_READY
        jz coi_done
          cmp BUF_COUNT[bx],0               ;any chars in the buffer ?
          je coi_done
            push si
            mov si,BUF_CHAROUT[bx]          ;get character out of buffer
            mov al,BUF_BUFFER[bx+si]
            out SS_DATA,al                  ;Compupro data port
            inc si                          ;if past end of buffer go back to
            and si,BUF_LEN-1
            mov BUF_CHAROUT[bx],si          ;update next char out
            dec BUF_COUNT[bx]               ;one less char in buffer
            pop si
            cmp BUF_FWAIT[bx],0FFh          ;if process is flag waiting
            jne coi_done                    ;and buffer is half empty
              cmp BUF_COUNT[bx],BUF_LEN/2
              ja coi_done
                mov BUF_FWAIT[bx],0
                mov dl,BUF_FLAGN[bx]        ;then set the flag
                push cx ! push dx           ;?SETFLAG alters AX,BX,CX,DX
                callf int_setflag           ;all AX,BX already saved
                pop dx ! pop cx
coi_done:
        pop bx
        cli                                 ;reset the PIC's
        mov al,NS_EOI                       ;signal non-specific end of inter
```

```
        out MASTER_PIC_PORT,al          ;PIC ports on Compupro
        out SLAVE_PIC_PORT,al
        pop ax
        mov ss,crt0_out_ss              ;restore stack
        mov sp,crt0_out_sp
        dec @bh_inint                   ;reset interrupt count
        pop ds
        iret

DSEG

        extrn   @bh_inint:byte          ;in interrupt count in BIOS
                                        ;Kernel Data Header

;       console output interupt stack area

crt0_out_sp     rw      1
crt0_out_ss     rw      1
                rw      32
crt0_out_tos    rw      0

out_buf_desc    rb      1               ;flag number - set by CDB_INIT
                db      0               ;"flag waiting" variable
                dw      0               ;number of chars in buffer
                dw      0               ;next char to take out of buffe
                rb      BUF_LEN.        ;buffer
```

CHARACTER I/O ERROR MESSAGES

The BIOS Kernel and the BDOS define an error return only from the CDB_INIT routines. The BIOS must handle all other character I/O errors it encounters. You can display error messages and also ask the user about what be action should be taken. Usually the choices give the user are retrying the operation again, ignoring the error, or aborting the program causing the error. The P_TERM system call can be made to terminate the program encountering an error. However, an error detected by an interrupt service routine cannot abort the running program. A status line if available, is a preferable location to display errors, causing fewer conflicts with screen oriented applications.

If you display error messages on the main part of the console, you should check the File System Error Mode for the process encountering the character I/O error. If the Return Error Mode is set it can be assumed the application does not want the screen altered and you should display messages only for catastrophic errors. The File System Error Mode is described in the Programmer's Guide and is set by the F_ERRMODE system call. The File System Error Mode is a byte located at byte 46h relative to the process environment segment. The process environment segment is in register ES on entry to all of the CHARIO CDB routines. The currently running process environment segment is also found in the word location at offset 04Eh relative to the SYSDAT segment. (See Appedix C) If the process' File System

Error Mode byte is equal to 0FFh, the process is in Return Error Mode and most error messages should not be displayed.

End of Section 6

Section 7

BIOS Disk I/O

This section covers customization of the disk I/O routines in the CP/M-86 Plus BIOS. The material in this section is separated into four subsections in the order needed for implementation.

The first subsection presents the information to implement the basic disk I/O routines. The second subsection describes enhancements to these routines for multiple logical disks sharing the same physical disk, for automatic density and side selection, for detection of media changes, for skewed-format disks, and for memory disk implementations. The third subsection covers the data structures the BDOS uses for disk I/O buffering. Last, a short discussion of BIOS disk I/O error messages is given.

Because GENCPM automatically generates the disk I/O buffering data structures, they are a supplementary topic. However, understanding these data structures is helpful when tuning disk I/O perfomance by using differing numbers of data and directory buffers.

BASIC DISK I/O

A CP/M-86 Plus disk driver is a combination of code routines and data structures you write and define. Each drive has four code routines to perform disk initialization, type of media determination, disk reads, and disk writes. The parameters to the disk read and write routines are passed to the BIOS on the stack and are accessed using the IO Parameter Block. The Disk Parameter Block (DPB) data structure describes the physical characteristics of a drive and the Disk Parameter Header (DPH) data structure represents each of the logical drives A-P, implemented in the system.

The CP/M-86 Plus disk organization is discussed first since it is affected by the DPB definition.

Disk Organization

A CP/M-86 disk is divided into at least two and often three areas. The first N tracks can be reserved for the bootstrap loader and CPMLDR, which read the CPM3.SYS file into memory. These tracks are called the system tracks. This area is optional and is needed only if the disk boots the system. For example, a hard disk not used for bootstrap operations has no system tracks.

The second area is the directory and starts immediately after the system tracks. The directory area keeps the names, the disk data areas, time and date stamps, and attributes of files. It also keeps the directory label for the disk. The system implementor defines the size of the directory area, which becomes static after system

boot. The directory size limits the number of files that can be created on a specific drive. However, the larger the directory, the smaller the data region that can be allocated to a file.

The third area is the data region. This area contains the data associated with files and all unallocated disk space.

Figure 7-1 illustrates the organization of a CP/M-86 Plus disk:

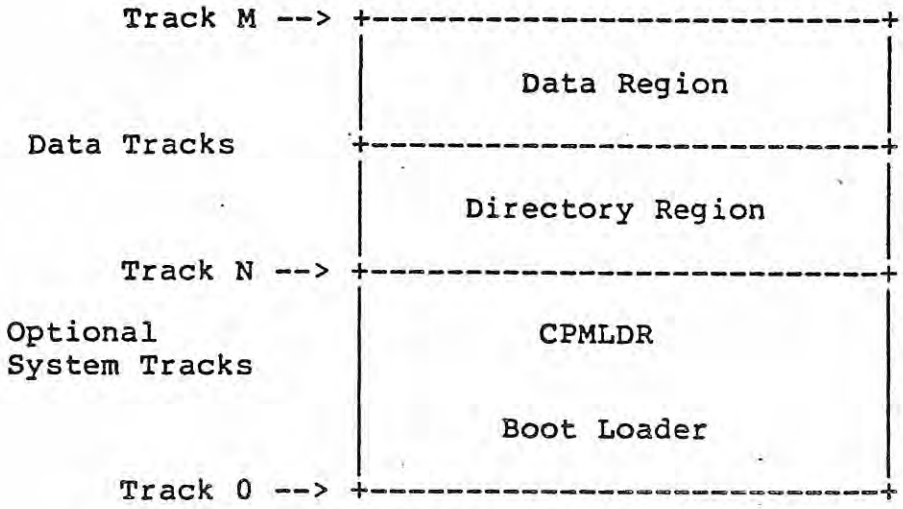


Figure 7-1. CP/M-86 Plus Disk Organization

In Figure 7-1, the first N tracks are the system tracks; CP/M-86 Plus uses the remaining tracks, the data tracks, for file directory and file data storage.

Note that eight-inch, single-density, IBM..3740-formatted diskettes should have two system tracks and a sector skewing of six to be compatible with other machines running CP/M with eight-inch single density drives. All CP/M-86 Plus disk accesses after system boot are directed to the data tracks of the disk.

Disk Parameter Block (DPB)

The physical characteristics of a drive are available to the BDOS via the Disk Parameter Block. Each different type of drive has a separate DPB, while physical drives with the same characteristics can share DPBs. For instance, machines with physically identical floppy drives can share the same DPB. Drives supporting different media types usually require one DPB per media type supported. The BDOS never changes any of the fields in the DPB, using it only as a information structure.

Listing 7-1, contained in the file DISK.LIB on the distribution

diskette, defines the DPB format.

Listing 7-1. Disk Parameter Block Format

```
;*****************************************************
;
;       Disk Parameter Block Equates
;
;*****************************************************
;
;       +-----+-----+-----+-----+-----+-----+-----+-----+
;  00h  |     SPT     | BSH | BLM | EXM |     DSM     |  DRM..
;       +-----+-----+-----+-----+-----+-----+-----+-----+
;  08h  ..DRM | AL0 | AL1 |     CKS     |     OFF     | PSH |
;       +-----+-----+-----+-----+-----+-----+-----+-----+
;  10h  | PHM |
;       +-----+

DPB_SPT         equ     word ptr 0
DPB_BSH         equ     byte ptr 2
DPB_BLM         equ     byte ptr 3
DPB_EXM         equ     byte ptr 4
DPB_DSM         equ     word ptr 5
DPB_DRM         equ     word ptr 7
DPB_AL0         equ     byte ptr 9
DPB_AL1         equ     byte ptr 10
DPB_CKS         equ     word ptr 11
DPB_OFF         equ     word ptr 13
DPB_PSH         equ     byte ptr 15
DPB_PHM         equ     byte ptr 16
```

Listing 7-2 is an example DPB definition from the DISKIO.A86 for a single sided, single density 8" diskette.

Listing 7-2. Disk Parameter Block Definition

```
;         1944:  128 Byte Record Capacity
;          243:  Kilobyte Drive  Capacity
;           64:  32 Byte Directory Entries
;           64:  Checked Directory Entries
;          128:  128 Byte Records / Directory Entry
;            8:  128 Byte Records / Block
;            8:  128 Byte Records / Track
;            2:  Reserved  Tracks

dpbs1:                                  ;single density, single sided
                dw      26              ;sectors per track
                db      3               ;block shift
                db      7               ;block mask
                db      0               ;extent mask
                dw      S1DSM-1         ;disk size - 1
```

```
dw      64-1            ;directory size - 1
db      0C0h            ;alloc0 - 2 directory blocks
db      00h             ;alloc1
dw      8010h           ;checksum size - 64/4
dw      2               ;offset by 2 tracks
db      0               ;physical sector shift
db      0               ;physical sector mask
```

Table 7-1 describes each field of the Disk Parameter Block. Appendix D includes a worksheet to help you calculate the DPB values.

Table 7-1. Disk Parameter Block Data Fields

Data Field	Explanation
DPB_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 DPB_PSH and DPB_PHM, described later in this table.
DPB_BSH	[Allocation block shift factor] This value is used by the BDOS to calculate a block number, given a logical record number, by shifting the record number DPB_BSH bits to the right. DPB_BSH is determined by the allocation block size chosen for the disk drive.
DPB_BLM	[Allocation block mask] This value is used by the BDOS to calculate a logical record offset within a given block by masking the logical record number with DPB_BLM. The DPB_BLM is determined by the allocation block size.

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

Table 7-1. (continued)

Data Field	Explanation
DPB_EXM	[Extent mask] The extent mask determines the maximum number of 16Kb logical extents that is contained in a single directory entry. It is determined by the number and size of the allocation block size.

BLOCK SIZE	EXM < 256	EXM >= 256
1,024	0	INVALID
2,048	1	0
4,096	3	1
8,192	7	3
16,384	15	7

DPB_DSM — [Disk storage maximum] The disk storage maximum defines the total formatted storage capacity of the disk drive, expressed in allocation blocks. This equals the total number of allocation blocks for the drive, minus 1. DPB_DSM must be less than or equal to 7FFFh. If the disk uses 1024-byte blocks (BSH=3, BLM=7) DPB_DSM must be less than or equal to 0FFh.

DPB_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. This determines the maximum number of directory entries allowed on a disk drive.

BLOCK SIZE	DIRECTORY ENTRIES	MAXIMUM DRM
1,024	32 * reserved blocks	511
2,048	64 * reserved blocks	1,023
4,096	128 * reserved blocks	2,047
8,192	256 * reserved blocks	4,095
16,384	512 * reserved blocks	8,191

Table 7-1. (continued)

Data Field	Explanation
DPB_AL0 DPB_AL1	[Directory allocation vector] The directory allocation vector is a bit map that is used to initialize the memory allocation vector that is created when a disk drive is logged in. Each bit represents an allocation block being used for the directory. The fields are filled in beginning with the high-order bit of DPB_AL0. This value is determined by the value of DPB_DRM (size of the directory) desired.
DPB_CKS	[Checksum vector size] The checksum vector size determines the required length of the directory checksum vector indicated in the Disk Parameter Header, and the number of directory entries that the BDOS will checksum. 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 ensure the data integrity of the disk system. The high-order bit in the DPB_CKS field indicates a permanent drive and allows more optimized performance, in other words, delayed 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 DPH_DOPEN byte in the DPH is set to 0FFh, indicating that the operator has opened the drive door. Normally, the disk is treated like a permanent drive, allowing more optimal use. In this case, adding 8000h to the nominal DPB_CKS value indicates a removable-media drive with door-open interrupt capability.
DPB_OFF	[Track offset] The track offset is the number of reserved tracks at the beginning of the disk. DPB_OFF is equal to the 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.

Table 7-1. (continued)

Data Field	Explanation
DPB_PSH	[Physical record shift factor] The physical record shift factor is used by the BDOS to calculate the physical record number from the logical record number. The logical record number is shifted DPB_PSH bits to the right to calculate the physical record.
DPB_PHM	[Physical record mask] The physical record mask is used by the BDOS to calculate the logical record offset within a physical record by masking the logical record number with the DPB_PRM value.

SECTOR SIZE	PSH	PHM
128	0	0
256	1	1
512	2	3
1024	3	7
2048	4	15
4096	5	31

Disk Parameter Header

The drive table in the BIOS Kernel Data Header (@BH_DRIVETABLE) contains 16 words, which correspond with the logical drive letters A-P. These words contain offsets of Disk Parameter Headers or a 0 value if the drive is not supported. The BDOS uses the DPHs to access all the other data structures related to a particular drive. Each DPH must be unique; two logical drives cannot share the same DPH.

Listing 7-3 shows the format of the DPH and is part of the file DISK.LIB on the distribution diskette.

Listing 7-3. Disk Parameter Header Format

```
;*******************************************************
;
;       Disk Parameter Header Equates
;
;*******************************************************
;
;       +-----+-----+-----+-----+-----+-----+-----+-----+
;  00h  |    XLT    |     SCRATCH     | DOPEN|  SCRATCH  |
;       +-----+-----+-----+-----+-----+-----+-----+-----+
;  08h  |    DPB    |     CSV   |     ALV     |  DIRBCB   |
;       +-----+-----+-----+-----+-----+-----+-----+-----+
;  10h  |   DATBCB  |    HSHTBL |     INIT    |   LOGIN   |
;       +-----+-----+-----+-----+-----+-----+-----+-----+
;  18h  |    READ   |    WRITE  |UNIT |CHNNL |   FLAGS   |
;       +-----+-----+-----+-----+-----+-----+-----+-----+

DPH_XLT         equ     word ptr 0
DPH_DOPEN       equ     byte ptr 5
DPH_DPB         equ     word ptr 8
DPH_CSV         equ     word ptr 10
DPH_ALV         equ     word ptr 12
DPH_DIRBCB      equ     word ptr 14
DPH_DATBCB      equ     word ptr 16
DPH_HSHTBL      equ     word ptr 18
DPH_INIT        equ     word ptr 20
DPH_LOGIN       equ     word ptr 22
DPH_READ        equ     word ptr 24
DPH_WRITE       equ     word ptr 26
DPH_UNIT        equ     byte ptr 28
DPH_CHNNL       equ     byte ptr 29
DPH_FLAGS       equ     byte ptr 30
```

Listing 7-4 below, shows an example DPH definition from the DISKIO.A86 file on the distribution disk.

Listing 7-4. Disk Parameter Heading Definition

```
; floppy disk 0
@dpha           dw      xltd3           ;translate table
                db      0, 0, 0         ;scratch area
                db      0               ;door open flag
                db      0, 0            ;scratch area
                dw      dpbd6           ;disk paramater table
                dw      0FFFFh, 0FFFFh  ;checksum, allocation vector
                dw      0FFFFh          ;directory bcb
                dw      0FFFFh          ;data bcb
                dw      0FFFFh          ;hash table
```

```
dw      fd_init         ;init routine
dw      fd_login        ;login routine
dw      fd_read         ;read routine
dw      fd_write        ;write routine
db      0               ;unit
db      0               ;channel 0
db      1               ;one flag used
```

Table 7-2 describes the fields in the DPH:

Table 7-2. Disk Parameter Header Data Fields

Data Field	Explanation
DPH_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 DPH_XLT to 0. 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.
SCRATCH	[Scratch area] The 5 bytes of zeroes 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 7-2. (continued)

Data Field	Explanation
DPH_DOPEN	[Door open flag] If the BIOS can detect that the drive door has been opened, it can set this flag to 0FFh when it detects that the operator has opened the door. It must also set the global door open flag, @BH_GDOPEN in the BIOS Header, to 0FFh at the same time. If @BH_GDOPEN is set to 0FFh, the BDOS then checks for a media change before performing the next file operation on that drive. The BDOS resets the @BH_GDOPEN flag when checked, as well as any of the DPH_DOPEN fields checked. Note the BDOS checks this flag only when a file related system call is initiated within the BDOS. DPH_DOPEN is not checked again until the next file related system call is made. Usually, this flag is only useful in systems that support door-open interrupts. If the BDOS determines that the drive contains a new diskette, the BDOS relogs-in the drive and resets the DPH_DOPEN field to 0. **Note:** If a door open interrupt is available, using this flag improves disk performance by as much as 30%, making the BDOS treat a removable-media drive as if it were a permanent drive. See the description of the DPB_CKS field in Table 7-1.
DPH_DPB	[Disk parameter block address] The DPH_DPB field contains the address of a Disk Parameter Block that describes the characteristics of the disk drive.
DPH_CSV	This field contains the offset of the Checksum Vector, 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. One byte must be in the Checksum Vector (CSV) for every four directory entries (or 128 bytes of directory). In short, Length(CSV) = (DPB_DRM/4)+1 (see the DPB Worksheet in Appendix D). If DPH_CKS in the DPB is 0 or 8000h, no checksum area is used, and DPH_CSV can be zero. Values for DPB_DRM and DPH_CKS are also calculated as part of the DPB Worksheet. If this field is initialized to 0FFFFh, GENCPM automatically creates the appropriate checksum vector structure

within SYSDAT and initializes the DPH_CSV field.

Table 7-2. (continued)

Data Field	Explanation
DPH_ALV	This field contains the offset of the Allocation Vector (ALV). The BDOS uses the ALV to track disk-storage allocation information. The allocation vector must be different for each DPH. The allocation vector is actually two separate vectors. One vector reflects the allocated blocks as recorded in the drive's directory; the second vector records the currently allocated blocks not yet recorded in drive's directory. Each vector contains one bit per each allocation block on the disk, rounded up to the nearest byte. The length of the ALV is double the length of one these allocation vectors or the Length(ALV) = (DPB_DSM/4)+2. Calculate the value of DPB_DSM as part of the DPB Worksheet provided in Appendix D. If this field is initialized to 0FFFFh, GENCPM automatically creates the appropriate data structures in the SYSDAT Table Area.
DPH_DIRBCB	This field contains the offset of the Directory Buffer Control Block (DIRBCB) Header. The DIRBCB Header contains the directory the offset of the first of the linked Directory Buffer Control Blocks for this drive. (See "Disk I/O Buffering" later in this section.) The BDOS uses directory buffers for all accesses of the disk directory. Several DPHs can refer to the same DIRBCB Header, or each DPH can reference an independent DIRBCB Header. If this field is 0FFFFh, GENCPM initializes the DPH_DIRBCB field and automatically creates the DIRBCB Header, the DIRBCBs, and the Directory Buffers for the drive, within SYSDAT.
DPH_DATBCB	This field contains the offset of the Data Buffer Control Block Header (DATBCDB) Address. The DATBCB Header contains the offset of the linked data buffers for this drive. (See "Disk I/O Buffering" later in this section.) 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, GENCPM initializes the DPH_DATBCB field, automatically creates the DATBCB Header

and DATBCBs within SYSDAT, and allocates space for the Data Buffers.

Table 7-2. (continued)

Data Field	Explanation
DPH_HSTBL	This field contains the paragraph address of the optional directory Hash Table (HSTBL) associated with a logical drive. The BDOS assumes the Hash Table offset address to be zero. If you decide not to use directory hashing, set DPH_HSTBL to zero. However, 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*(DPB_DRM+1) bytes, where DPB_DRM = length of the directory - 1. In other words, each entry in the hash table must hold four bytes for each directory entry of the disk. If this field is 0FFFFh, GENCPM initializes DPH_HSTBL and automatically creates the appropriate Hash Table.
DPH_INIT	This is the offset of the first-time initialization code for the drive. The BIOSINIT routine in the BIOSKRNL module calls each DPH's DPH_INIT routine during system initialization. DPH_INIT can perform any necessary hardware initialization, such as setting up the controller and interrupt vectors, if any. Upon entry, register BX contains the offset of the DPH for this drive.
DPH_LOGIN	This is the offset of the login routine for the drive. The DPH_LOGIN routine is called before the BDOS reads the directory the first time to log in the drive. The BDOS logs in a drive by reading the directory and computing the drive free space and other values. If the information is available through the hardware, DPH_LOGIN allows the automatic determination of the media type.
DPH_READ	This is the offset of the sector read routine for the drive. When the DPH_READ routine is called, the base address of the IOPB (see "IOPB Data Structure" after this table) is contained in register BP. The parameters necessary for the read operation are all contained in the IOPB.

Table 7-2. (continued)

Data Field	Explanation
DPH_WRITE	This is the offset of the sector write routine for the drive. When the DPH_WRITE routine is called, the address of the IOBP is contained in register BP. The parameters necessary for the write operation are all contained in the IOPB (see "IOPB Data Structure" after this table).
DPH_UNIT	The DPH_UNIT byte contains the drive code, relative to the disk controller, for the disk drive referenced by this DPH. For instance if a disk controller supports logical drives C: and E:, the DPH_UNIT fields in DPHC and DPHE are set to two and four respectively. Only the BIOS uses this field.
DPH_CHNNL	The DPH_CHNNL byte contains the ID of the controller that supports this device. For instance, if a one disk controller handles logical drives A: and B: while a second controller manages logical drives C: and D:, the DPH_CHNNL field is set to zero in DPHA and DPHB. Since drives C: and D: use the second controller, the DPH_CHNNL fields in DPHC and DPHD are set to one. Only the BIOS uses this field.
DPH_FLAGS	The DPH_FLAGS byte contains the number of system flags used by this drive. If more than one drive share a controller, then the first DPH for that controller should indicate the number of flags used; all other DPHs for drives that share the controller should have a zero in their DPH_FLAGS fields. The first DPH of several that share a controller can be identified by a DPH_CHNNL value of zero. GENCPM uses DPH_FLAGS in calculating the minimum number of system flags to allocate.

IOPB Data Structure

The disk Input/Output Parameter Block (IOPB) contains the parameters required for the IO_READ and IO_WRITE function calls in the BIOS Kernel and the DPH_READ and DPH_WRITE functions in the DISKIO modules you supply. The IOPB is located on the stack when the BDOS

calls the BIOSENTRY routine in the BIOS Kernel. The IOPB structure uses the BP register since indirect addressing using the BP register of the 8086/8088 processors is relative to the SS (stack segment) register. The IOPB is defined relative to the value BP as set by the READ_WRITE routine in the Kernel. BP obviously cannot be modified by the disk I/O routines if the IOPB is going to be used.

DPH_READ and DPH_WRITE can index or modify IOPB parameters directly on the stack, since they are removed by the BDOS after the BIOS IO_READ or IO_WRITE functions return.

Listing 7-5 shows the format of the IOPB. This information is also found in the file DISK.LIB on the distribution disk. Table 7-7 discusses each field in the IOPB.

Listing 7-5. Input/Output Parameter Block (IOPB)

```
;*******************************************************
;
;       Input/Output Parameter Block Definition
;
;*******************************************************
;
;  Read and Write disk parameter equates
;
;  At the disk read and write entries,
;   all disk I/O parameters are on the stack
;   and the stack at these entry points is as
;   follows:
;
;           +-------+-------+
;       +14 |  DRV  | MCNT  |   Drive and Multi sector 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
;           +---------------+
;      BP+0 |    RET_ADR    |   Local ENTRY return address
;           +---------------+    (assumes one level of call
;                                 from ENTRY routine)
;
;These parameters (except for the return addresses)
;may be indexed and modified directly on the stack;
;they are removed on return to the BDOS.
```

```
iopb_mcnt       equ     byte ptr 15[bp]
iopb_drive      equ     byte ptr 14[bp]
iopb_track      equ     word ptr 12[bp]
iopb_sector     equ     word ptr 10[bp]
iopb_dmaseg     equ     word ptr 8[bp]
iopb_dmaoff     equ     word ptr 6[bp]
```

Table 7-3. IOPB Data Fields

Field	Explanation
IOPB_DRV	[Logical drive number] The logical drive number specifies the logical disk drive on which to perform the DPH_READ or DPH_WRITE operation. The drive number can range from 0 to 15, corresponding to drives A through P respectively.
IOPB_MSCNT	To transfer logically consecutive physical disk sectors to or from contiguous memory locations, the BDOS issues an IO_READ or IO_WRITE function call with IOPB_MSCNT set greater than 1. This allows the BIOS 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 BIOS can transfer up to 16Kb directly to or from the DMA address in a single operation. Note, the IOPB_MSCNT is distinct from the Multisector Count set by the F_MULTISEC system call. The F_MULTISEC system call sets a logical (128 byte sector) Multisector Count for file I/O transfers between the transient and the BDOS. For a more complete explanation of multisector operations, along with example code and suggestions for implementation within the BIOS, see "Skewed Multisector Disk I/O" later in this section.

Table 7-3. (continued)

Field	Explanation
IOPB_TRACK	The IOPB_TRACK defines the track for the specified drive to seek. The BDOS defines IOPB_TRACK relative to zero. For disk hardware which defines track numbers beginning with a physical track of one, your DPH_READ and DPH_WRITE routines must increment the track number before passing it to the disk controller. The BDOS uses the values you define in the DPB to calculate IOPB_TRACK. Usually the DPB is defined to directly correspond to the physical disk and the IOPB_TRACK value is the physical track number. However tracks can be defined to include both sides of a double-sided drive or a cylinder of a multiplatter drive. When a track is defined by the DPB to be more than one physical track, the BIOS calculates the head from the IOPB_SECTOR number.
IOPB_SECTOR	The IOPB_SECTOR defines the sector for a read or write operation on the specified drive. The BDOS defines the IOPB_SECTOR relative to zero, so for disk hardware which defines sector numbers beginning with a physical sector of one, the DPH_READ and DPH_WRITE routines increment the sector number before passing it to the disk controller. The sector size is determined by the parameters DPB_PSH and DPB_PHM defined in the Disk Parameter Block. Usually the DPB is defined so the sector size is equal to the physical sector size of the disk. If the specified drive uses a skewed-sector format, the DPH_READ and DPH_WRITE routines must translate the sector number according to the translation table specified in the Disk Parameter Header.

Table 7-3. (continued)

Field	Explanation
IOPB_DMAOFF, IOPB_DMASEG	The DMA offset and segment define the address of the disk data transfer buffer for the read or write operation. This DMA address can reside anywhere in the one-megabyte address space of the 8086/8088 microprocessor. If the disk controller for the specified drive can only transfer data to and from a limited range of addresses, DPH_READ or DPH_WRITE must block-move the data between the DMA address and a similar buffer located in the area accessible to the controller before a write or following a read operation.
IOPB_RETSEG, IOPB_RETOFF	These two words are used to return to the BDOS from the BIOSENTRY routine and must be preserved through DPH_READ or DPH_WRITE.
IOPB_LOCALRET	The local return address returns to the BIOSENTRY routine in the BIOS Kernel when the DPH_READ or DPH_WRITE routines finish.

DPH_DISK I/O Routines

This section discusses the CP/M-86 Plus BIOS hardware-dependent disk functions you supply. The BIOS Kernel accesses these functions through their offsets contained in the DPH fields DPH_INIT, DPH_LOGIN, DPH_READ, and DPH_WRITE. There must be a valid routine for each of the four functions in every DPH in the BIOS; the DPH_INIT, DPH_LOGIN, DPH_READ, and DPH_WRITE fields cannot be zero.

Table 7-4. DPH_Disk I/O Routines

Routine Explanation

DPH_INIT

The DPH_INIT routine for each disk device initializes the device hardware during system initialization at the request of the BIOSINIT routine. BIOSINIT calls the DPH_INIT routine for each disk device represented by a DPH, using the address in the DPH_INIT field of the DPH. A DPH_INIT routine can simply return if the initialization is performed by another DPH_INIT routine. This occurs when several DPHs share the same disk controller.

Entry Registers: BX = address of DPH
DS = SYSDAT (BIOS data segment)
ES = process environment

Exit Registers: BX,DS, ES preserved

DPH_LOGIN

The DPH_LOGIN routine can optionally determine the current media type in a removable media drive. The BIOS Kernel calls DPH_LOGIN when the BDOS calls the Kernel IO_SELECT routine and indicates a "first time" select. First time selects occur only when the drive is first accessed and after the DPH_READ or DPH_WRITE routines signal a media change to the BDOS. The register conventions for the call to DPH_LOGIN from the BIOS Kernel are the following:

Entry Registers: BX = offset of DPH
DS = SYSDAT (BIOS data segment)
ES = process environment

Exit Registers: BX, DS, ES preserved

The DPH_LOGIN function call allows the BIOS to determine density, the number of sides, and any other disk parameters that can change during operation. Once the new parameters are determined, the hardware might need to reinitialized. If the type of drive changes, the DPH_DPB field is changed to point the DPB defining the new drive. "Auto Density/Side Selection," which appears later in this section, discusses the DPH_LOGIN routine in more detail.

Table 7-4. (continued)

Routine Explanation

DPH_READ, DPH_WRITE

The CP/M-86 Plus BDOS performs disk I/O with a single BIOS call to the BIOS Kernel IO_READ or IO_WRITE functions using the parameters contained in the IOPB. The BIOS Kernel, in turn, calls the OEM- written disk routines DPH_READ and DPH_WRITE, which perform the disk operations.

If a physical error occurs during a DPH_READ or DPH_WRITE operation, the function should perform several retries (ten is recommended) to attempt to recover from the error before returning an error condition.

The following are the register conventions for DPH_READ and DPH_WRITE:

Entry Registers:
BX = offset of DPH
BP = offset of IOPB on stack
DS = SYSDAT (BIOS data segment)
ES = process environment

Exit Registers:
AL = 0 if no error
AL = 1 if physical error
AL = 2 if read-only disk
AL = 0FFh if media density has changed
DS, ES preserved

If the IOPB_MSCNT field is equal to one, DPH_READ and DPH_WRITE routines transfer the single physical sector specified in the IOPB. If a physical error occurs, DPH_READ and DPH_WRITE return 1h in AL after attempting retries. DPH_WRITE can additionally return AL equal to 2 if a drive is physically read-only.

For drives supporting several types of media, DPH_READ and DPH_WRITE should return an 0FFh in AL if the BIOS detects a change in media density. After returning an 0FFh, the BDOS calls the IO_SELECT routine in the Kernel, which in turn calls the DPH_LOGIN routine for the same drive. See "Automatic Density and Side Selection" later in this Section.

If the IOPB_MSCNT is greater than 1, the DPH_READ or DPH_WRITE routines transfer the specified number

of physical sectors before returning to the BIOS Kernel. The DPH_READ and DPH_WRITE routines transfers as many physical sectors as the specified drive's disk controller can handle in one operation.

Table 7-4. (continued)

Routine	Explanation
DPH_READ, DPH_WRITE	(continued)

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 transfer or write, a 01H is returned in AL.

If the disk controller hardware for the specified drive does not have a feature for making multisector transfers, DPH_READ and DPH_WRITE can make the number of single physical-sector transfers defined by the IOPB_MSCNT. Making mulitiple single physical-sector transfers is recommended when first bringing up the disk I/O routines, unless you already have multisector I/O routines working from another implementation. DPH_READ and DPH_WRITE must increment the sector number and add the number of bytes in each physical sector to the IOPB_DMAOFF address for each successive single physical-sector transfer

The BDOS initializes the IOPB_DMAOFF and IOPB_DMASEG such that a multisector transfer will not cause the value of IOPB_DMAOFF to overflow. If, during a multisector transfer, the sector number exceeds the number of the last physical sector of the current track, DPH_READ and DPH_WRITE routines increment IOPB_TRACK and reset IOPB_SECTOR to zero.

Listing 7-6 below, shows a simple implementation of a Multi-sector read/write routine that performs single sector operations until all the sectors are transfered. The DISKIO.A86 module on the distribution disk contains a read/write routine that performs Multi-sector transfers at the controller level. The RW.A86 file provides another example showing a Multi-sector read/write routine that cannot transfer across a 64Kbyte page boundary because of hardware restrictions.

In Listing 7-6, if IOPB_MSCNT is zero, the routine returns with an error. Otherwise, it calls the read/write routine (IOHOST:) for the present sector specified by the current values of IOPB_TRACK and IOPB_SECTOR. IOHOST puts a return code in the RETCODE variable, which is tested after IOHOST returns. If there is no error, the IOPB_MSCNT value is decremented. When IOPB_MSCNT equals zero, the read or write is finished and the routine returns. If not, the sector number to read or

write is incremented. If, however the sector number now exceeds the number of sectors on a track (MAXSEC) the IOPB_TRACK number is incremented and the IOPB_SECTOR number is set to zero. Then the routine performs the number of reads or writes remaining to equal IOPB_MSCNT, each time adding the size of a physical sector to IOPB_DMAOFF. Listing 7-6 illustrates multisector operations assuming a disk controller only supporting single sector I/O.

Listing 7-6. Multisector I/O

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

hd_io:
        push es                 ;save process environment
        cmp iopb_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 iopb_mcnt         ;decrement multisector count
          jz return_rw          ;if mcnt = 0 return
            mov ax,iopb_sector
            inc ax              ;next sector.
            cmp ax,maxsec
            jb same_trak        ;is sector < max sector
              inc iopb_track    ; no - next track
              xor ax,ax         ; initialize sector to 0
same_trak:
            mov iopb_sector,ax  ;save sector #
            add iopb_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 process environment
        ret                     ;return with error code in AL

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

iohost:
 ...
 ...
 ...

        ret
```

DISK I/O ENHANCEMENTS

You can modify the CP/M-86 Plus disk I/O system in several ways. A large hard disk can be divided into several logical drives to provide a more convenient file organization. Door open interrupt can be detected to increase disk I/O and improve data integity. The automatic detection of media types prevents the user from having to invoke a utility that informs the BIOS of the current media type. This is helpful when a removable media drive supports single- and double-density media. Skewed disk formats are often necessary for compatiblity with media written from other machines or operating systems. **Multiple Logical Drives**

A large nonremovable-media storage device, such as a hard disk, can be divided into several logical drives for user convenience. This is done using the DPB_OFF (track offset) field in the DPB.

The DPB_OFF field can define the beginning of a logical drive as shown:

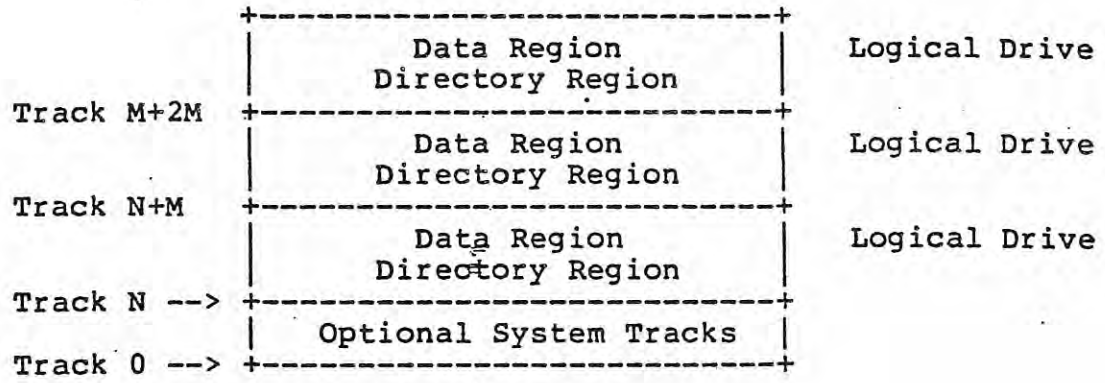


Figure 7-2. Multiple Logical Drives

Figure 7-2 shows three logical drives mapped onto one physical drive. Three separate DPHs and DPHs are required for each drive. Even if the logical drives are identical in size three different DPBs are necessary since the DPB_OFF is different for each drive and is set to N, N+M and N+2M.

Detecting Media Changes

All disk drives under CP/M-86 Plus are classified as either permanent or removable. In general, removable drives support media changes; permanent drives do not. The discussion in this subsection considers media changes when the media type is preserved. The next subsection treats the detection of different media types, such as single- or double-density formatted diskettes.

The CP/M-86 Plus file system distinguishes between permanent and removable drives. If a drive is permanent, the BDOS always accepts the contents of physical record buffers as valid. In addition, it also accepts the results of hash table searches on the drive.

On removable drives, the validity of the physical sector buffers is conditional in order to protect against writing to a drive whose media has changed. The BDOS logs in removable media drives by computing and storing checksums and hash codes for the drive's directory. The checksums for a particular drive are stored in the checksum vector whose offset resides in the DPH_CSV field. The hash codes are stored in the hash table whose offset resides in the DPH_HSTBL field. (These fields and data areas are usually set and allocated automatically by GENCPM.)

Before the BDOS performs certain directory-related functions, it verifies the disk has not changed. The BDOS does this by computing checksums for the parts the disk directory being used and comparing them with the corresponding checksums previously computed. If the checksums differ, the operation is denied.

A similar situation occurs with directory hashing on removable drives. When an unsuccessful hash table search occurs, the BDOS attempts to locate the directory entry by reading the directory. During this pass through the directory, the checksums are computed and compared with the ones stored in the checksum vector.

When the checksum values do not match, the BDOS assumes the media has changed. The BDOS logs out the drive by invalidating its directory and data buffers, then again attempts to log in the disk which forces the entire directory to be read.

The net result of these actions is that there is a significant performance penalty associated with removable drives as compared to permanent drives. In addition, the protection provided by classifying a drive as removable is not complete. Media changes are only detected during directory operations. If the media is changed while writing file data when no directory accesses are required, the new disk will be damaged.

Another option for supporting drives with removable media is available if an interrupt can be generated when the drive door is opened. This option allows the drive to be treated as permanent media by the BDOS until the occurance of a door open interrupt. If your hardware provides this support, you can increase disk I/O performance up to 30% and improve the integrity of removable media by the following procedure:

- Compute the normal DPB_CKS value for a removable media drive. This is the size of the check sum vector and is equal to the total number of directory entries, divided by four. Then set the most significant bit in the DPB_CKS by adding the value of 8000h to the DPB_CKS field. For example, set the CKS field for a disk with 96 (60h) directory entries to 8018H. This bit signals the BDOS to treat the drive specially.

- Implement an interrupt service routine that sets the @BH_GDOPEN byte to 0FFh in the BIOS Kernel Data Header and the DPH_DOPEN byte to 0FFh for the drive that signaled the door open condition.

The BDOS checks @BH_GDOPEN on certain disk related function calls. If @BH_GDOPEN is equal to 0, it implies that no drive doors have been opened in the system. If @BH_GDOPEN is set to 0FFh, the BDOS checks the DPH_DOPEN byte of each currently logged-in drive. If the DPH_DOPEN byte is 0FFh, the BDOS reads the entire directory on that drive and computes and compares checksums. Any directory buffers for this drive are temporarily ignored, forcing the verifying directory reads to the disk. If the checksums match, it is assumed the door was opened but the media was not changed. If the checksums differ, the drive is logged out, then logged in again as required.

Automatic Density and Side Selection

Some physical drives can support several different kinds of media. For example, floppy disk drives and controllers can often accept several densities formatted on one or two sides of the diskette. If the BIOS can detect the media type, automatic density and side selection can be implemented. Automatic selection of the media type in the BIOS replaces the need for a special transient program written by the system implementor. This transient must be invoked by the end-user each time the media type is changed.

To support auto density and side selection, the DPH_READ and DPH_WRITE routines must be able to determine when the media has changed. Also, the BIOS must be able to determine the media type.

To implement auto density support, a DPB is included in the BIOS for each media type expected, or routines to alter DPB values to reflect the media type currently being used. When the DPH_READ or DPH_WRITE routines detect a media change they must return AL equal to 0FFh back to the BDOS. The BDOS then makes a "first time" select call to the BIOS Kernel IO_SELDSK function. In turn, the IO_SELDSK function calls the DPH_LOGIN routine for the drive. The DPH_LOGIN function, supplied by the system implementor, determines the media type and sets the DPH_DPB field to the offset of the DPB that describes the media.

If unable to determine the format, the DPH_LOGIN function can return a 0, indicating that the select operation was not successful. The IO_SELECT function returns the error and the BDOS prints a message or returns an error to the application depending on the BDOS error mode. (See F_ERRMODE system call in the Programmer's Guide). Table

7-4 shows the DPH_LOGIN register conventions.

Once the DRV_LOGIN routine has determined the format of the disk, the BDOS assumes this format is correct and uses the DPB currently associated with the drive for subsequent read and write operations.

Skewed Multisector Disk I/O

CP/M-86 Plus supports multiple physical sector read and write operations at the BIOS level to minimize rotational latency on block disk transfers. Multisector I/O is implemented in the BIOS 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 BIOS read/write function routine translate each logical sector number into a physical sector number. Then it creates a table (Figure 7-3) of DMA addresses with each sector's DMA address indexed into the table by the physical sector number.

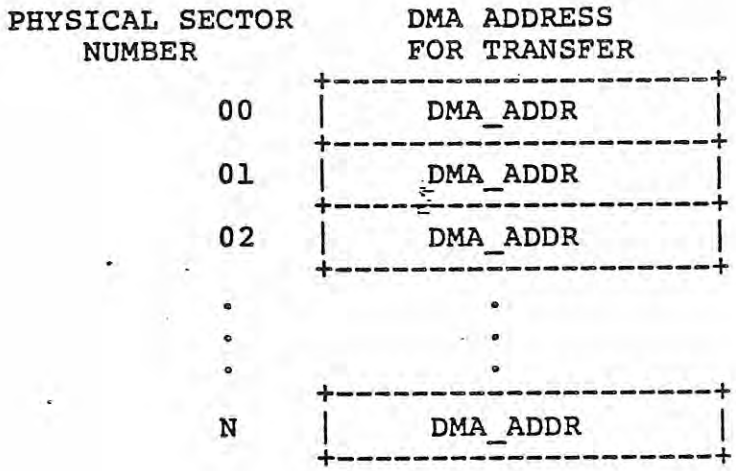


Figure 7-3. DMA Address Table for Skewed Multisector I/O

As a result, the requested sectors are sorted into the order in which they physically appear on the track. Often the required sectors on the track to be transferred in one disk rotation. As a sector is read or written, it is transferred to or from its proper DMA address.

During a multisector data transfer, if the sector number exceeds the number of the last physical sector of the current track, the BIOS increments IOPB_TRACK and resets the IOPB_SECTOR to zero. It can then complete the operation for the balance of sectors specified in the DPH_READ or DPH_WRITE function call.

Listing 7-7 illustrates multisector I/O for a skewed disk. The routine gets the DPH address by calling the IO_SELDSK function. It checks to verify a non-zero DPH address, and returns if the address is invalid (zero). Next the disk parameters are taken from the DPH and DPB to be stored in local variables. Once the physical record size is computed from the 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 physical sectors are numbered relative to one for that particular drive.

The DMA table (DMATBL) is filled with the DMA addresses for the requested sectors on the current track. The RW_SECTS routine transfers the sectors to the proper DMA addresses and returns to READ_WRITE if more sectors are to be read on the next track. The READ_WRITE routine continues to calculate the DMA addresses on succeeding tracks and transfer the sectors by calling RW_SECTS until all requested sectors are transferred.

In this example, local values that begin with "I" (such as ISECTOR and ITRACK) are initialized by RW_SECTS and are parameters used by the read or write routine whose offset is in register SI.

The following code fragment illustrates multisector unskewing. It is assumed that this fragment is called from the DPH_READ and DPH_WRITE routines you supply with the registers set as indicated.

Listing 7-7. Skewed Multisector Disk I/O

```
;*******************************************************
;*
;*      DISK I/O CODE SEGMENT
;*
;*******************************************************

rw_skew:        ;unskews and reads or writes multi sectors
;-------
;       entry:  SI = read or write routine address
;               BX = DPH
;               BP = IOPB
;               DS = SYSDAT
;               ES = process environment
;       exit:   AL = return code
;               DS and ES preserved

ret_error:
          mov al,1                  ; return error if not
          ret
dsk_ok:
        mov ax,DPH_XLT[bx]
        mov xltbl,ax                ;save translation table address
        mov bx,DPH_DPB[bx]
        mov ax,DPB_SPT[bx]
        mov maxsec,ax               ;save maximum sector per track
        mov cl,DPB_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
rw_1:
          mov ax,iopb_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 iopb_track          ; next track
            xor ax,ax
            mov iopb_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
```

Listing 7-7. (continued)

```
no_trans:
          xor bh,bh
          mov bl,al                ;sector # is used as the index
          shl bx,1                 ; into the dma offset table
          mov ax,iopb_dmaoff
          mov dmatbl[bx],ax        ;save dma offset in table
          add ax,secsiz            ;increment dma offset by the
          mov iopb_dmaoff,ax       ; physical sector size
          inc iopb_sector          ;next sector
          dec iopb_mcnt            ;decrement multi sector count
          jnz rw_1                 ;if mcnt <> 0 store next sector dma
rw_secta:                          ;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,iopb_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 occured return
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 process environment
        push ds! pop es
        rep stosw                  ;initialize table to 0ffffh
        pop es                     ;restore process environment
        ret
```

Listing 7-7. (continued)

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

xltbl   dw      0       ;translation table address
maxsec  dw      0       ;max sectors per track
secsiz  dw      0       ;sector size
dmatbl  rw      50      ;DMA address table
```

Memory Disk Implementation

In CP/M-86 Plus, a disk drive is any I/O device that has a directory and is capable of reading and writing data in sectors up to 4 Kb in size. The BIOS can therefore treat a wide variety of peripherals as disk drives if desired. A memory disk is an example of this flexibility.

A memory disk (M: disk) uses an area of RAM to simulate a disk drive, making a very fast temporary disk. GENCPM can specify the M: disk as the temporary drive. This section discusses the M: disk implementation as shown in Listing 7-8.

In Listing 7-8, the M: disk memory space begins at the 0C000h paragraph boundary and extends for 128 Kb through the 0DFFFh paragraph. The BIOSINIT routine calls the DPH_INIT routine in DPHM, which initializes the directory area of the M: disk, the first 16 Kb to 0E5h. 0E5h's signify unused directory entries to the BDOS.

Both the M: disk DPHM_READ and DPHM_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 M: disk sector size is defined by the DPB to be 128 bytes making the calculation of a paragraph address simply a shift operation. The IOPB_MSCNT (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 (DPB) for the M: disk, shown at the end of the example, is provided for reference. A hash table can be provided for the M: disk Disk Parameter Header in order to further

increase performance. (GENCPM is usually used to automatically create the hash table.)

Listing 7-8 illustrates an M: disk implementation:

Listing 7-8. Example M: Disk Implementation

```
mdiskbase       equ     0C000h  ;base paragraph
                                ;address of mdisk
CSEG

dphm_init:      ;initialize M: disk RAM directory area
;---------
        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

dphm_login:     ;no media change possible for M: disk
;----------
;       entry:  BX = DPH
;       exit:   BX = DPH

        ret

dphm_read:      ;read from M: disk
;---------
;       entry:  BX = DPH
;               IOPB on stack
;       exit:   none

; Reads the sectors specified by the IOPB
; to the DMA address also specified in the IOPB.

        call mdisk_calc         ;calculate byte address
        push es                 ;save process environment
        les di,dword ptr iopb_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 process environment
        xor ax,ax               ;return with good return code
        ret

dphm_write:     ;write to M: disk
;----------
;       entry:  BX = DPH
;               IOPB on stack
;       exit:   none
```

Listing 7-8. (continued)

```
; Write the sectors specified in the IOPB
; to the DMA address also specified in the IOPB

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

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

        mov bx,iopb_track       ;pickup track number
        mov cl,3                ;times eight for sector relative
        shl bx,cl               ;to beginning of M: disk
        mov cx,iopb_sector      ;plus IOPB_SECTOR number
        add bx,cx               ;gives relative sector number to transfer
        mov cl,3                ;times eight for paragraph relative numbe
        shl bx,cl               ;of starting sector to transfer
        add bx,mdiskbase        ;plus base address of M: disk
        mov cx,64               ;length in words for 1 sector move
        mov al,mcnt
        xor ah,ah
        mul cx                  ;length * multisector count
        mov cx,ax
        cld
        ret

DSEG

dpbm    rb      0               ;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
```

DISK I/O BUFFERING

Directory and file data is buffered in physical sectors within in the BIOS. (A physical sector is the sector size as defined by the DPB for the drive.) Since GENCPM generates the the data structures and initializes the fields in the DPH for disk buffering, this material is optional.

The BDOS uses BCBs to locate and manage physical sector buffers. A Buffer Control Block (BCB) describes each physical sector buffer. Directory BCBs (DIRBCBs) describe directory buffers and Data BCBs (DATBCBs) describe file data buffers. The BCBs are linked together to describe multiple buffers with directory and data BCBs kept on separate lists.

Each logical drive has directory and data buffers associated with it via the Disk Parameter Header (DPH) representing the drive. The DPH fields DPH_DIRBCB and DPH_DATBCD contain the offsets of BCB Headers. The BCB Header is a three-byte structure that contains the offset of the first of the linked BCBs. Several logical drives as represented by different DPHs can specify the same list of BCBs.

Each BCB has a BCB_LINK field containing the address of the next BCB in the list, or 0 if it is the last BCB. All BCB Headers and BCBs must reside within the SYSDAT segment.

Listing 7-9 is an example BCB Header definition:

Listing 7-9. BCB Header Definition

```
bcb_head        dw      dirbcb0         ;first DIRBCB
                db      0ffh
```

The first word of the BCB Header, as previously mentioned, contains the offset of the first BCB in a list of BCBs. The third byte in the BCB Header is used by the BDOS and must be initialized to 0FFh.

Directory Buffer Control Block

The Directory Buffer Control Block (DIRBCB) is used by the BDOS to manage disk directory buffers in the BIOS. The buffer associated with the BCB must be large enough to accomodate the largest physical sector associated with any drive using the BCBs.

Listing 7-10 shows the DIRBCB format:

Listing 7-10. Directory Buffer Control Block (DIRBCB) Format

```
;*********************************
;*
;*   DIRBCB Format
;*
;*********************************

;       +------+------+------+------+------+------+------+------+
; 00H:  | DRV  |       RECORD       | WFLG | 00H  |    TRACK    |
;       +------+------+------+------+------+------+------+------+
; 08H:  |   SECTOR    |    BUFOFF   |     LINK    |   RESERVED  |
;       +------+------+------+------+------+------+------+------+

BCB_DRV         equ     byte ptr 0
BCB_RECORD      equ     byte ptr 1
BCB_WFLG        equ     byte ptr 4
BCB_SEQ         equ     byte ptr 5
BCB_TRACK       equ     word ptr 6
BCB_SECTOR      equ     word ptr 8
BCB_BUFOFF      equ     word ptr 10
BCB_LINK        equ     word ptr 12
```

Listing 7-11 illustrates a DIRBCB definition:

Listing 7-11. DIRBCB Definition

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

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

Table 7-5 defines the DIRBCB fields:

Table 7-5. DIRBCB Data Fields

Data Field	Explanation
BCB_DRV	This field is the logical drive number that identifies the disk drive associated with the physical sector contained in the buffer. The initial value of the BCB_DRV must be 0FFh. If BCB_DRV = 0FFh, then the BDOS considers the buffer available for use. The BDOS initializes the other BCB fields when a BCB and its buffer are used by the BDOS.
BCB_RECORD	The BCB_RECORD number identifies the logical record position of the current buffer for the specified drive. The BCB_RECORD number is relative to the beginning of the logical disk, where the first record of the directory is logical record number zero. (Logical records are 128 bytes long.)
BCB_WFLG	[Write pending flag] The BDOS sets the BCB_WFLG to 0FFh to indicate the buffer contains unwritten data. When the data is written to disk, the BDOS sets the BCB_WFLG to zero.
00h	Reserved for system use.
BCB_TRACK	The BCB_TRACK is the track number associated with the BCB's buffer. The BCB_TRACK number is the calculated by the BDOS from drive's DPB values.
BCB_SECTOR	BCB_SECTOR is the sector number associated with the BCB's buffer. The BCB_SECTOR number is calculated by the BDOS from the drive's DPB. The BCB_SECTOR is usually the physical sector number.
BCB_BUFOFF	For DIRBCBs, this field equals the offset address of the buffer within SYSDAT.

Table 7-5. (continued)

Data Field	Explanation
BCB_LINK	The BCB_LINK field contains the offset address of the next BCB in the linked list, or zero, if this is the last BCB.
BCB_RESERVED	Reserved for system use.

The BCB_DRV field is the logical drive the buffer is associated with or is set to 0FFh indicating the buffer is unallocated. The initial value of the BCB_DRV field must be 0FFh.

When the BCB_WFLG field equals 0FFh, the buffer contains data that the BDOS has to write to the disk before the buffer is available for other data.

For file system integrity, the data and directory BCBs must be separate. Since directory buffers are never "write pending" having separate directory buffers ensures that a buffer is available when the BDOS reads the directory to detect media changes. If data and directory buffers were mixed, all of the buffers could contain "write pending" data and the directory could not be read prior to a write.

Data Buffer Control Block

Listing 7-11 shows the format of the Data Buffer Control Block (DATBCB):

Listing 7-11. Data Buffer Control Block (DATBCB)

```
;*********************************
;*
;*   DATBCB Format
;*
;*********************************

;        +------+------+------+------+------+------+------+------+
; 00H:   | DRV  |       RECORD       | WFLG | 00H  |    TRACK    |
;        +------+------+------+------+------+------+------+------+
; 08H:   |   SECTOR    |    BUFSEG   |    LINK     |   RESERVED  |
;        +------+------+------+------+------+------+------+------+
```

Listing 7-11. (continued)

```
bcb_drv     equ     byte ptr 0
bcb_record  equ     byte ptr 1
bcb_wflg    equ     byte ptr 4
bcb_seq     equ     byte ptr 5
bcb_track   equ     word ptr 6
bcb_sector  equ     word ptr 8
bcb_bufseg  equ     word ptr 10
bcb_link    equ     word ptr 12
```

The DATBCB is identical to the DIRBCB, except for the BCB_BUFSEG field.

BCB_BUTSEG equals the segment address of the Data Buffer. The offset of the buffer is assumed to be zero. The data buffer can not share memory with transient program area (TPA) and must be on a segment (paragraph) boundary.

DPH_HSTBL and BCB_BUFSEG Initialization

The hash table address for a particular logical drive is a segment value kept in the DPH_INIT field. The address of the data buffer associated with a DIRBCB is also a segment value. If you define the hash tables or the directory buffers in the BIOS you must "fix-up" these addresses to be segment values. The following code fragment accomplishes this. Note GENCPM automatically sets the segment address of the hash tables and data buffers it creates in the appropriate DPH_HSTBL and BCB_BUFSEG fields.

Listing 7-12. DPH_HSHTBL and BCB_BUFSEG Initialization

```
; Initialize DPH_HSTBL and BCB_BUFSEG fields.  The hash
; table and data buffers must be paragraph aligned.  This
; code fragment fixes up the offset values in DPH_HSTBL and
; BCB_BUFSEG to be segment values.  This code must be
; executed during BIOS initialization if GENCPM is not used
; to generate the hash tables or the data buffers.  This code
; assumes NONE of the hash tables or data buffers are created
; by GENCPM.  If you use GENCPM to generate some of the hash
; tables and data buffers then this code must be modified to
; only fix up the appropriate structures and not those
; already set to segment addresses by GENCPM.  This code
; also assumes all of the data buffers in the BIOS are shared
; with drive A:'s and thus only A:'s are fixed up.

        mov cx,16                       ;16 maximum drives
        xor si,si                       ;SI = 0
hash_init:
        push cx                         ;save drive count
```

```
        mov bx,@bh_dphtable[si]         ;BX = DPH address
        test bx,bx
        jz next_dph                     ;if not 0, BX = DPH
          mov ax,DPH_HSTBL[bx]          ;AX = hash table offset
          or ax,ax ! jz next_dph        ;if 0, no hash table
            mov cl,4                    ;compute paragraphs from
            shr ax,cl                   ;start of SYSDAT
            mov dx,ds                   ;add SYSDAT segment
            add ax,dx                   ;AX = hash table segment
            mov DPH_HSTBL[bx],ax        ;make the fixup
next_dph:
        pop cx                          ;restore the drive count
        loop hash_init

;       Initialize data BCB segment addresses
;       all drives share the same set of data buffers

        mov bx,offset @bh_dph_table
        mov ax,DPH_DATBCB[si]           ;AX=DATBCB header
        mov ax,[si]                     ;AX=DATBCB
next_datbcb:
        mov ax,BCB_BUFSEG[si]           ;AX=data buffer offset
        mov cl,4                        ;calculate paragraphs from
        shr ax,cl                       ;SYSDAT
        mov dx,ds                       ;add in SYSDAT to get
        add ax,dx                       ;absolute segment address
        mov BCB_BUFSEG[si],ax           ;make fixup
        mov si,BCB_LINK[si]             ;SI=next BCB
        or si,si                        ;0 if end of linked list
        jnz next_datbcb
```

DISK I/O ERROR MESSAGES

The BIOS Kernel and the BDOS define error returns from the DPH_READ, DPH_WRITE, and DPH_LOGIN routines. The DPH_INIT routine has no error return defined. When an error is returned from the DISKIO module, the BDOS displays error messages on the console unless the program encountering the error is in Return Error Mode. (See the F_ERRMODE system call in the Programmer's Guide.) If a physical error (AL=1) is returned from DPH_READ and DPH_WRITE the BDOS displays the following message:

```
CP/M ERROR on d:  Disk Read/Write Error
BDOS Function = xx File = filespec
```

(d: one of the logical drives A:-P:, xx is the last BDOS function the program encountering the error made with an INT 224 operation, and filespec is the file name and type.)

The DPH_WRITE routine can also return a Read/Only disk error (AL=2) that results in the following BDOS message:

```
CP/M ERROR on d:  Read-Only Disk
BDOS Function = xx File = filespec
```

The Read-Only Disk error can also be returned if an attempt to write to a drive set to Read-Only through the DRV_SETRO system call. If DPH_LOGIN routine returns an error (BX=0) the following BDOS message is displayed:

```
CP/M ERROR on d:  Invalid Drive
BDOS Function = xx File = filespec
```

Appendix H discusses the BDOS error interrcept that allows you to modify or translate these and other BDOS error messages.

If you plan to display more information about a specific hardware error the discussion from Section 6, "Character I/O Error Messages" applies here also. As stated in Section 6 if you display error messages on the main part of the console, you should check the File System Error Mode for the process encountering the character I/O error. If the Return Error Mode is set it can be assumed the application does not want the screen altered and you should display messages only for catastrophic errors. The File System Error Mode is a byte located at byte 46h relative to the process environment segment. The process environment segment is in register ES on entry to all of the DISKIO DPH_ routines. The currently running process environment segment is also found in the word location at offset 04Eh relative to the SYSDAT segment. (See Appedix C) If the process' File System Error Mode byte is equal to 0FFh, the process is in Return Error Mode and most error messages should not be displayed.

End of Section 7

Section 8

Clock Support

This section discusses the functions provided by the CP/M-86 Plus CLOCK module. The CLOCK module must perform clock hardware initialization and provide a periodic "system tick" interrupt for dispatching and maintaining the time and date variables within the SYSDAT segment.

TICK INTERRUPT ROUTINE

The tick interrupt is used primarily to generate dispatches that force compute-bound processes to relinquish the CPU so that other processes can run. The system tick rate, which you define, determines the dispatch frequency for compute-bound processes. The recommended tick unit is 16.67 milliseconds, corresponding to a tick 60 times a second or 60 Hertz. When operating on 50-Hertz power, use a unit of 20 milliseconds if it is more convenient. The @BH_TICKSEC field in BIOS Data Header must be set to the number of ticks per second to permit accurate use of the P_DELAY system call.

For CP/M-86 Plus to run more than one program at a time, the tick interrupt service routine must execute a JMPF (Jump Far instruction) to INT_DISPATCH. When the JMPF to INT_DISPATCH is made, the DS register on entry to the interrupt service routine must be on the stack. INT_DISPATCH is a double word located in the SYSDAT segment that points to the dispatcher within the BDOS. The dispatcher saves the environment of the running process and restores the environment of the next process ready to run. If there is no other process to run, the dispatcher performs a POP DS instruction and an IRET (Interrupt Return instruction) back to the interrupted process. The changing of process environments by BDOS dispatcher is also referred to as "context switching."

Once every system tick, the system tick flag (system flag #1) must be set by the tick interrupt if the @BH_DELAY field in the BIOS Kernel Data Header is set to 0FFh. The BDOS sets @BH_DELAY to 0FFh when a process makes a P_DELAY system call. @BH_DELAY is set to 0 by the BDOS when no processes are delaying.

The tick interrupt routine must also, once per second, update the system time of day structure. The time of day structure is kept in the SYSDAT segment and is shown in Appendix C. The BDOS accesses this structure for file time and date stamping and for the system calls that set and return the time and date.

For systems with a time of day and calendar chip, the clock interrupt service routine must ensure the SYSDAT time of day variables correspond to the chip's time of day. If a date and calendar chip is part of the hardware, the DATE utility can need to be replaced by your own utility to set the time of day and date on the chip.

EXAMPLE TICK INTERRUPT

The following tick interrupt listing is similar to the one contained in the example BIOS' CLOCK.A86 file on the distribution disks. The equates for the Programmable Interrupt Controller and the SYSDAT variables are from the the files PIC.LIB and SYSDAT.LIB also on the distribution disks. The tick interrupt in the example BIOS is generated by a counter timer chip approximately 60 times a second. The tick interrupts are counted by the interrupt service routine to determine time periods of a second, a minute and an hour. The time of day variables in the SYSDAT segment are updated accordingly. In this example the tick intervals are not exactly 1/60 of a second, so the number of ticks counted in a second is switched between 60 and 61 for better accuracy over long periods of time.

Section 4 discusses the general structure of an interrupt service routine under CP/M-86 Plus. Note the TICK_INT routine below uses the @BH_ININT (in interrupt count) since other interrupt service routines in the example BIOS reenable interrupts.

Listing 8-1. Tick Interrupt Service Routine

```
                                        ;equates for 8259A
NS_EOI          equ     20h             ;non specific end of interrupt
MASTER_PIC_PORT equ     50h
SLAVE_PIC_PORT  equ     52h

int_dispatch    equ     dword ptr .34h  ;exit from interrupt handler
int_setflag     equ     dword ptr .38h  ;interrupt SETFLAG function

tod_day         equ     word ptr .5Fh   ;number of days since 1/1/78
tod_hr          equ     byte ptr .61h   ;current hour in packed BCD
tod_min         equ     byte ptr .62h   ;current minute in packed BCD
tod_sec         equ     byte ptr .63h   ;current second in packed BCD

CSEG
        extrn   ?waitflag:near          ;BIOS Kernel routines
        extrn   ?dispatch:near
        extrn   @sysdat:word            ;system and BIOS data segment

;*****************************************************
;
;       Tick Interrupt Service Routine
;
;**********      ****************************************

tick_int:
;========
        push ds ! mov ds,cs:@sysdat
        inc @bh_inint                   ;signal executing interrupt handle
        mov saveax,ax
```

```
        dec tick_cnt                    ;tick count
        jnz cont_tick                   ;if second not yet up, branch to
          mov al,last_cnt               ;get previous tick count
          xor al,1                      ;toggle low order bit
          mov last_cnt,al
          mov tick_cnt,al               ;TICK_CNT = either 60 or 61

          mov al,tod_sec                ;TOC_SEC is packed BCD
          inc al ! daa                  ;keep it packed BCD
          mov tod_sec,al
          cmp al,60h ! jb cont_tick     ;compare with 60h BCD
            mov tod_sec,0
            mov al,tod_min              ;TOD_MIN is packed BCD
            inc al ! daa                ;keep it packed BCD
            mov tod_min,al
            cmp al,60h ! jb cont_tick   ;compare with 60h BCD
              mov tod_min,0
              mov al,tod_hr             ;TOD_HR is packed BCD
              inc al ! daa              ;keep it packed BCD
              mov tod_hr,al
              cmp al,24h ! jb cont_tick ;compare with 24h BCD
                mov tod_hr,0
                inc tod_day             ;TOD_DAY is a binary value
cont_tick:
        cmp @bh_delay,0FFh              ;are any processes delaying
        jne not_delaying                ;via the P_DELAY system call ?
          mov tick_ssreg,ss             ;switch to local stack
          mov tick_spreg,sp             ;for CALLF to INT_SETFLAG
          mov ss,@sysdat                ;BIOS data segment
          mov sp,offset tick_tos        ;set to tick interrupt stack

          push bx ! push cx             ;registers used by INT_SETFLAG
          push dx                       ;AX already saved in SAVEAX
          mov dl,1                      ;system flag #1 is the tick flag
          callf int_setflag             ;set it
          pop dx ! pop cx ! pop bx

          mov ss,tick_ssreg             ;restore stack
          mov sp,tick_spreg
not_delaying:
        mov al,NS_EOI                   ;signal PIC's interrupting
        out MASTER_PIC_PORT,al          ;condition has been
        out SLAVE_PIC_PORT,al           ;satisfied

        mov ax,saveax                   ;restore AX
        dec @bh_inint ! jz tick_exit
                                        ;go back to incompleted
                                        ;interrupt service routine
tick_exit:
        jmpf int_dispatch               ;if more than one process
                                        ;is ready to run, give the
                                        ;CPU to another ready process
```

```
;*********************************************************
;
;       Clock Data Segment
;
;*********************************************************
        DSEG

        extrn   @bh_inint:byte          ;variables in BIOS Kernel
        extrn   @bh_delay:byte          ;Data Header

                rw      15
tick_tos        rw      0               ;tick interrupt stack
tick_ssreg      rw      1               ;save registers
tick_spreg      rw      1               ;during tick interrupt
saveax          rw      1               ;here

last_cnt        db      61              ;adjust for counter
tick_cnt        db      61              ;timer chip's tick frequency
```

End of Section 8

Section 9

System Generation

This section describes the procedures necessary to generate the CP/M-86 Plus system contained in the CPM3.SYS file.

Generation of the CPM3.SYS file is a four-stage process. First, you must assemble all BIOS modules into OBJ format files using RASM-86. (OBJ refers to Intel Object Module Format.) Next, use the MODEDIT utility to examine each BIOS module OBJ files to resolve any mulitple CDB or DPH symbol definitions. Third, use LINK-86 to link all of the OBJ format BIOS modules together to create the BIOS3.SYS file. Finally, use GENCPM to create the system image file CPM3.SYS from the BIOS3.SYS, BDOS3.SYS, and optionally, the CCP.CMD files.

ASSEMBLING THE BIOS MODULES

The Programmer's Utilities Guide documents RASM-86. Use the following commands to assemble the example BIOS modules found on the distribution diskette:

```
A>RASM86 BIOSKRNL
A>RASM86 INIT
A>RASM86 CHARIO
A>RASM86 DISKIO
A>RASM86 CLOCK
```

The assembly of these modules results in the OBJ format files BIOSKRNL.OBJ, INIT.OBJ, CHARIO.OBJ, FDISKIO.OBJ, and CLOCK.OBJ. If you are debugging a particular module, use the RASM-86 $LO to cause local symbols to be included in the symbol file generated by LINK-86. The RASM-86 $NC (no case) option, which prevents RASM-86 from automatically translating symbol names to uppercase, should not be used to generate the BIOS modules. This is because the MODEDIT utility searches for specific symbol names in uppercase.

MODEDIT UTILITY

MODEDIT is an OBJ file editor that resolves conflicts between CDB and DPH publ nd external declarations. It takes the following command lin :

```
MODEDIT [...]el Mod1 [ Mod2 Mod3 Mod4 ... ]
```

MODEDIT allows new device drivers to be added to an exiting BIOS when the names of the CDBs and DPHs in the existing BIOS are not known by the writer of the device driver. Both the BIOS and the new device drivers must be in OBJ format. MODEDIT modifies the files specified on the command line and creates no new output files.

The CDB and DPH labels in the BIOSKRNL and the other BIOS modules you supply must take the form @CDBX and @DPHX, where X is an ASCII character A through P.

MODEDIT scans the first file specified on the command line for the external symbols with the names @CDBA through @CDBP and @DPHA through @DPHP. This first file must be the BIOS Kernel or at least contain the BIOS Kernel Data Header. The rest of the files specified are other BIOS modules containing public declarations for CDBs and DPHs. There can be as many as 16 public CDB declarations and as many as 16 public DPH declarations. BIOS modules that do not contain public declarations of CDBs or DPHs need not be modified by MODEDIT.

MODEDIT relabels CDB and DPH public symbols in the OBJ files on the command line in the order in which the OBJ files are specified on the command line from left to right. If more than one @CDBX or @DPHX public symbol occurs within an OBJ file, the names are assigned in the order of appearance within the file.

For example, consider the following command:

```
A>MODEDIT BIOSKRNL, CHAR1, CHAR2, DISK1, DISK2
```

Assume the public declarations for the symbols @CDBE and @CDBD appear in the file CHAR1.OBJ in the same order as they appear in this sentence. MODEDIT changes these symbol names to @CDBA and @CDBB respectively. If CHAR2.OBJ contains the public symbols in the order @CDBB then @CDBA, MODEDIT renames the symbols to @CDBC and @CDBD respectively. This occurs because @CDBA and @CDBB were used for the first two Character Device Blocks in the CHAR1.OBJ file. MODEDIT handles DPH symbols similarly.

LINKING THE BIOS MODULES

After you use MODEDIT to rename the CDBs and DPHs, link the separate BIOS OBJ modules to form the BIOS3.SYS file. Use LINK-86 to link these example BIOS modules. The Programmer's Utilities Guide describes LINK-86 in greater detail. The following is an example of a LINK-86 command line:

to be reduced

```
A>LINK86 BIOS3.SYS = BIOSKRNL,INIT,CHARIO,DISKIO,CLOCK,ZERO.L86 [DATA[ORI
                                                    N[ØF00], SCAR
```

Since the BIOS data starts at 0F00h, the ORIGIN option must be present in the LINK-86 command when creating the BIOS3.SYS file. The use of the ORIGIN option assumes there are no ORG statements or only "ORG 0" statements in the BIOS modules. When there is no ORG statement in a module, RASM-86 assumes an ORG of 0. The BIOSKRNL must be the first OBJ file in the LINK-86 command line since the BIOS Kernel Data Header and the BIOS Kernel Code Header must start the data and code groups (CMD format) in the BIOS3.SYS file. The order of the other BIOS modules does not usually matter, except the ZERO.L86 module must be last. ZERO.L86 is supplied on the distribution disks and is a library file containing public

definitions for the symbols @CDBA through @CDBP and @DPHA through @DPHP. If these symbols are not defined in one of the BIOS modules you supply, LINK-86 and ZERO.L86 force their definitions to a zero value.

LINK-86 takes the following form:

```
A>LINK86 BIOS3.SYS = BIOSKRNL,MOD1,MOD2,...MODN,ZERO.L86 [DATA[ORIGIN[OF0
                                                          ], SEARCH]
```

The OBJ files labeled MOD1,MOD2,...MODN are replaced by the names of the BIOS modules you supply. If, for example, another module called HDISKIO for hard disk support is to be added to the example BIOS, LINK-86 would take the following form:

```
A>LINK86 BIOS3.SYS = BIOSKRNL,INIT,CHARIO,DISKIO,HDISKIO,CLOCK,ZERO.L86 [I
                                                    TA[ORIGIN[OF##]], SEA
```

The LINK-86 INPUT option is helpful when you are continually generating a BIOS3.SYS file during development. The INPUT option allows the command tail to be read from a file. For instance, if the command tail is placed into the file BIOS.INP, the LINK-86 command becomes the following:

```
A>LINK86 BIOS[I]
```

GENCPM UTILITY

You can use the GENCPM utility to create the operating system memory image contained in the file CPM3.SYS. This file becomes the memory resident part of the CP/M-86 Plus operating system. You must read CPM3.SYS into memory at a specific location, then transfer control to it. GENCPM runs under either CP/M-86 1.X or Concurrent CP/M.

GENCPM builds the CPM3.SYS file from the files BDOS3.SYS, BIOS3.SYS, and optionally, CCP.CMD. Use GENCPM to allocate and create several data structures needed by the BIOS. These structures are the disk buffers, buffer control blocks, disk allocation vectors, disk checksum vectors, and disk hash tables. GENCPM can also reserve extra memory for the BIOS.

During BIOS development, CPM3.SYS is usually read into memory for debugging by DDT-86 or SID-86. After BIOS development, use CPMLDR to load CPM3.SYS. Subsequent sections in this guide cover the CPMLDR and debugging the BIOS.

The following paragraphs explain how to invoke and respond to the questions of GENCPM. The items in parentheses that are a part of GENCPM questions are default values. A default-value numeric is hexidecimal unless it is preceded by a pound sign (#), which indicates the numeric is decimal. You can answer any question either in hexidecimal or decimal. Four-digit (16-bit) values, such as 0108, are displayed and accepted as input by GENCPM in paragraph units (16 bytes). These paragraph values are memory addresses or memory lengths.

Invoke GENCPM by using one of these command lines:

```
A>GENCPM
A>GENCPM [AUTO]
```

The first command runs GENCPM interactively, causing a series of questions you must answer to be displayed. The [AUTO] option (abbreviated to [A]) allows GENCPM to run without console input and is useful as part of a submit file to generate the CPM3.SYS file. When you use the [AUTO] option, answers to the questions normally displayed by GENCPM are read from the file GENCPM.DAT. You can also use the GENCPM.DAT file to supply the default answers to the GENCPM questions when GENCPM is run interactively. GENCPM.DAT is an ASCII file that you can create by using an editor, or by using a GENCPM option.

GENCPM displays one Main Menu and several different screens of questions. Each of the different screens GENCPM displays appears in this section as a figure, followed by an explanation for each screen. Note the default values shown in the GENCPM screens used in this section were choosen for tutorial purposes and cannot be used to generate a working CPM3.SYS file from the example BIOS on the distribution disks.

In the following discussion, a question that can be answered in the GENCPM.DAT file is referred to as a question variable. GENCPM searches the GENCPM.DAT file for the question variable keywords and the associated answer. A line in the GENCPM.DAT file takes the following general form, in which value equals the answer for that question:

```
Question Variable = value <CR>
```

The easiest way to create a GENCPM.DAT file is to have GENCPM do it for you by responding with a Y to the "Create a new GENCPM.DAT file" question in Figure 9-1. If modifications are needed, edit the file directly or run GENCPM again to generate another GENCPM.DAT. The end of this section shows an example GENCPM.DAT file.

GENCPM Initial Questions

Figure 9-1 shows the initial questions displayed by GENCPM. The answers to these questions configure GENCPM each time it is run and are not part of the CPM3.SYS file.

```
CP/M-86 Plus System Generation
Copyright (C) 1983, Digital Research, Inc.
==========================================

Use GENCPM.DAT file for defaults                (Y)?

Clear screen sequence                           (1B,45)?
Home cursor sequence                            (1B,48)?

Accept new GENCPM parameters                    (Y)?
```

Figure 9-1. GENCPM Initial Questions Screen

The following are the questions asked by the initial screen:

Use GENCPM.DAT file for defaults (Y)?

Enter Y - GENCPM gets its default values from the file GENCPM.DAT. Default values are displayed in parentheses to the left of the ?. If you simply press <CR> after a GENCPM question, the default value is the answer to the question.

Enter N - GENCPM uses the built-in default values. The GENCPM utility has its own set of defaults "built-in" to the GENCPM.CMD file that are used in this case.

Note that this question does not appear if no GENCPM.DAT file currently exists on the default drive and user.

No question variable is associated with this question.

Clear screen sequence (1B,45)?

Enter the clear screen character sequence for this terminal. The values shown here are the hex ASCII codes for ESC and E. Values must be separated by commas. You may want to answer this question with a null (0) when using the CTRL-P function to echo GENCPM's console output to a printer. (Decimal values can be entered by preceding them with a # for GENCPM questions expecting a numeric reply.)

Question Variable: CLRSCR

Home cursor sequence (1B,48)?

Enter the character sequence for moving the cursor to the home position on the terminal. Values must be separated by commas. The values shown here are the ASCII codes for ESC and H. You may want to answer this question with a null (0) when using the

CTRL-P function to echo GENCPM's console output to a printer.

Question Variable: HOMSCR

Accept new GENCPM Parameters (Y)?

Enter Y - GENCPM proceeds to the main menu. GENCPM is configured for this session and you are ready to start generating the CPM3.SYS file. These initial questions cannot be repeated after this reply.

Enter N - GENCPM repeats the previous questions and displays your previous input in the parentheses so you can change any mistakes.

No question variable is associated with this question.

GENCPM System Generation Main Menu

GENCPM displays the Main Menu screen after you exit the initial menu. The Main Menu gives options for GENCPM help, four sub-menus, and two different ways of terminating the GENCPM session. The questions asked by GENCPM are divided into several catagories, each of which is represented by a sub-menu. Figure 9-2 shows the Main Menu:

```
CP/M-86 Plus System Generation
Copyright (C) 1983, Digital Research, Inc.
==========================================

CP/M-86 Plus GENCPM System Generation Main Menu
-----------------------------------------------

 1.  GENCPM Help.
 2.  Display/Change GENCPM Parameters.
 3.  Display/Change System Parameters.
 4.  Display/Change Memory Allocation Parameters.
 5.  Display/Change Disk Buffer Allocation.
 6.  Generate a system and exit.
 7.  Exit without generating a system.

Enter Number:
```

Figure 9-2. GENCPM System Generation Main Menu

Enter Number:

The Main Menu requests one of the option numbers be entered. No question variable is associated with this question.

When you finish with the help option or one of the sub-menus, you are returned to the Main Menu. When you change a value in the sub-menus, the new values appear in parentheses and become the default values for this session of GENCPM. Thus, if you select a sub-menu again, your previous answers appear in parentheses.

Option 1: GENCPM Help

Selecting option 1 from the Main Menu displays the following screen:

```
                    GENCPM HELP
                    -----------

GENCPM lets you edit and generate a system image
from operating system modules on the default disk
drive. A detailed explanation of each GENCPM
parameter may be found in Section 9 of the
CP/M-86 Plus Installation Guide.

GENCPM assumes the default values shown within
parentheses. All numbers are in hexadecimal
unless preceded with "#", indicating a decimal
value. All four digit values are in paragraph
units. To change a parameter, enter the new value
and type <CR>.

Press RETURN to return to the main menu.
```

Figure 9-3. GENCPM Help Screen

The help display asks no questions and has no associated question variables.

Option 2: Display/Change GENCPM Parameters

Selecting option 2 of the Main Menu causes the GENCPM Parameter Screen to appear. This screen queries for file-related information.

```
CP/M-86 Plus GENCPM Parameter Setup
-----------------------------------

Create a new GENCPM.DAT file                                    (N)?
Destination drive                                               (C:)?
Delete (instead of rename) old CPM3.SYS file                    (N)?
Permanently attach the CCP to the operating system              (N)?

Accept new GENCPM parameters                                    (Y)?
```

Figure 9-4. GENCPM Parameter Screen

The following explains each GENCPM Parameter Screen question:

Create a new GENCPM.DAT file (N)?

Enter N - GENCPM does not create a new GENCPM.DAT file.

Enter Y - If option 6 (Generate a system and exit) of the Main Menu is selected to exit GENCPM, a new GENCPM.DAT file is created.

Question Variable: CRTDATF

Destination drive (A:)

Enter the drive letter on which the CPM3.SYS file is to be created. If you want to use the default drive, and you are using the GENCPM.DAT file for defaults, remove the DESTDRV line in GENCPM.DAT. The CPM3.SYS is placed in the current default user area of the destination drive.

Question Variable: DESTDRV

Delete (instead of rename to CPM3.OLD) CPM3.SYS file (N)?

Enter N - GENCPM renames the existing CPM3.SYS file to CPM3.OLD.

Enter Y - GENCPM deletes the existing CPM3.SYS file and creates a new CPM3.SYS file.

Question Variable: DELSYS

Permanently attach the CCP to the Operating System (N)?

Enter Y - GENCPM includes the CCP.CMD file found on the current default drive and default user in the operating system image. When the resulting CP/M-86 Plus system is booted up, it does not need a CCP.CMD file on disk, though the memory area occupied by the operating system is larger.

Enter N - GENCPM does not attach the CCP to the operating system. A CCP.CMD file must exist on the initial default drive when the system is run.

Question Variable: CCPYES

Accept new GENCPM Parameters (Y)?

Enter Y - GENCPM returns to the Main Menu.

Enter N - GENCPM repeats the previous questions and displays your replies as the defaults. You can modify your earlier answers if a mistake was made.

No question variable is associated with this question.

Option 3: Display/Change System Parameters

Selecting option 3 of the Main Menu results in the following screen. The answers to this screen affect internal variables within CP/M-86 Plus, change the memory location of CP/M-86 Plus, reserve space for extra system flags, and allocate extra buffer space for the BIOS.

```
CP/M-86 Plus GENCPM System Parameter Setup
------------------------------------------

Backspace echoes erased character                (N)?
Rubout echoes erased character                   (N)?
Number of console columns                        (#80)?
Number of lines in console page                  (#24)?
Initial default drive                            (A:)?
Ticks per second                                 (#60)?
Number of additional flags                       (#0)?
Base of CP/M-86 Plus                             (0040)?
Data Base of CP/M-86 Plus                        (0000)?
Amount of space reserved in OS data segment      (0000)?

Accept new system definition                     (Y)?
```

Figure 9-5. GENCPM System Parameters Screen

The following explains the questions in the System Parameters Screen:

Backspace echoes erased character (N)?

This question only affects the behavior of the C_READBUF system call. The backspace character (CTRL-H, 08h) deletes a character from the buffer when using the C_READBUF system call.

Enter N - A backspace moves the cursor back one column and erases the character at the new cursor position.

Enter Y - A backspace prints the deleted character, then moves the cursor forward one column.

Question Variable: BACKSPC

Rubout echoes erased character (Y)?

This question only affects the behavior of the C_READBUF system call. The rubout character (RUB, 7Fh) deletes a character from the buffer when using the C_READBUF system call.

Enter N - A rubout moves the cursor back one column and erases the character at the new cursor position.

Enter Y - A rubout prints the deleted character, then moves the cursor forward one column.

Question Variable: RUBOUT

Number of console columns (#80)?

Enter the number of columns (characters-per-line) for your console. The answer to this question is accessible to transient programs through the S_SYSVAR system call.

The C_READBUF system call uses the answer to this question for line editing. A character in the last column should not force a new line for console editing in CP/M-86 Plus. If your terminal does force a new line automatically, enter the number of columns minus one.

Question Variable: PAGWID

Number of lines in console page (#24)?

Enter the number of lines per screen for your console. The answer to this question is used by transients to prompt before scrolling information off the screen. It is accessible through the S_SYSVAR system call.

Question Variable: PAGELEN

Initial default drive (A:)?

Enter the drive letter the prompt is to display after booting up the system. This drive is not "logged in", when the system first boots up, unless the CCP must be read off it.

Question Variable: BOOTDRV

Ticks per second (#60)?

Enter the number of ticks per second the system clock generates. GENCPM sets the @BH_TICKSEC field in the BIOS Kernel Data Header using the answer to this question. The BIOS ?CLOCKINIT routine can also change this field. It is accessible to transient programs through the S_SYSVAR system call.

Question Variable: TICKS

Number of additional flags (#0)?

Enter the number of additional system flags to be used. GENCPM allocates the number of flags requested in all the CDBs and DPHs found in the BIOS3.SYS file, plus the four flags reserved by CP/M-86 Plus for internal use. Additional flags requested here can be used by the BIOS for other devices, such as the tick interrupt service routine and by field installable device drivers.

Question Variable: ADDFLGS

Base of CP/M-86 Plus (0040)?

Enter the starting paragraph address of the operating system. This value is also the code segment of the BDOS.

Question Variable: OSBASE

Data base of CP/M-86 Plus (0000)?

Enter the paragraph address of the operating system data segment. Change the default value only if the operating system image is to be ROMed. See Appendix G for more information on ROMing CP/M-86 Plus.

Question Variable: OSDBASE

Amount of space reserved in OS data segment (0000)?

Enter the size in paragraphs of an uninitialized data buffer that is within the SYSDAT segment. Use the default value to allocate no memory. GENCPM sets @BH_BUFLEN in the BIOS Kernel Data Header to the number of paragraphs reserved, and places the offset of the reserved area in the @BH_BUFBASE field.

Question Variable: ADDMEM

Accept new system definition (Y)?

Enter Y - GENCPM returns to the Main Menu.

Enter N - GENCPM redisplays this menu with your previous answers as the new default values.

No question variable is associated with this question.

Option 4: Display/Change Memory Allocation Parameter.

Selecting option 4 from the Main Menu causes the following screen to display:

```
CP/M-86 Plus GENCPM Available Physical Memory Table Setup
---------------------------------------------------------

Partition  Base   Length
-------------------------
0       (0040,  1FC0)?
1       (2001,  0FFF)?
2       (0000,  0000)?
3       (0000,  0000)?
4       (0000,  0000)?
5       (0000,  0000)?
6       (0000,  0000)?
7       (0000,  0000)?

Accept new memory definitions  (Y)?
```

Figure 9-6. GENCPM Memory Allocation Parameters Screen

This screen requests the base and length of all available RAM, excluding the interrupt vector area in the lowest 1Kbyte of memory. The memory specified must include memory where the operating system is to be placed as determined by the "Base of CP/M-86 Plus" question in the System Parameter Screen (see Figure 9-5).

GENCPM trims the memory specified by this menu, reflecting the memory available for loading transient programs. This remaining memory is called the Transient Program Area (TPA). GENCPM initializes the Memory Descriptor table table in the BIOS Kernel Data Header (@BH_MEMDESC) to define the TPA memory. The BIOS INIT module can adjust the Memory Descriptor table according to the memory present on a particular machine. See Appendix F, "Memory Descriptor Format" and the example BIOS INIT module in the INIT.A86 file on the distribution disks.

The first partition shown in Figure 9-6 specifies memory from 40:0 to 1FFF:0, the second partition skips one paragraph and specifies memory from 2001:0 to 2FFF:0, inclusive. Because these two memory areas are noncontiguous, the BDOS cannot coalese them into one area. Physically contiguous memory is thus made logically noncontiguous, thereby preventing one transient program from allocating all memory with one memory allocation request. However, bear in mind that each separate memory area defined requires a Memory Descriptor and the total number of Memory Descriptors available to describe memory fragmentation during system operation is limited to 32.

Question Variable: MEMPART# (# = 0 to 7)

Option 5: Display/Change Disk Buffer Allocation

Selecting option 5 of the Main Menu causes the following screen to display. Use this screen to allocate directory and data buffers and hash tables for the drives defined in the BIOS3.SYS file.

```
CP/M-86 Plus GENCPM Disk Buffer Setup
-------------------------------------

Drive  Secsize Memory Allocated Dirbufs Databufs Hashing
-----  ------- ---------------- ------- -------- -------
 A:     0040H       034DH       (  08 ,    04 ,    Yes )?
 B:     0040H       0000H       (  A: ,    A: ,    Yes )?
 C:     0040H       0000H       (  A: ,    A: ,    Yes )?
 D:     0040H       034DH       (  03 ,    02 ,    Yes )?
 E:     0040H       0000H       (  04 ,    D: ,    Yes )?

    Accept new buffer definitions (Y)?
```

Figure 9-7. Disk Buffer Allocation Screen

GENCPM finds each defined DPH in the BIOS3.SYS file and displays its drive letter in this menu. GENCPM can only set the DPH_HSTBL, DPH_ALV, DPH_DATBCB, and DPH_DIRBCB fields if they are initialized in the BIOS to 0FFFFh. If they are not, GENCPM assumes each field is the offset of the appropriate data structure already defined

within the BIOS. "Disk Parameter Header (DPH)" in Section 7 explains how to manually calculate the memory needed for these DPH data structures.

If DPH_ALV is 0FFFFh, GENCPM calculates and reserves the amount of memory the drive requires for allocation vectors. DPH_ALV is set to the offset of allocation vectors.

DPH_HSTBL is handled similarly to DPH_ALV by GENCPM, except that directory hashing is optional. When hashing is selected for a drive, GENCPM reserves a separate hash table and places the paragraph address of the hash table in the corresponding DPH_HSTBL field. Directory hashing provides a substantial performance improvement and is encouraged.

When DPH_DATBCB and DPH_DIRBCB are set to 0FFFFh, GENCPM creates the number of buffers specified under the Dirbufs and Databufs headings. More directory buffers than data buffers are usually specified since directory buffers provide more performance benefit. GENCPM also creates the linked list of BCBs and a BCBROOT associated with the buffers.

GENCPM also allocates uninitialized buffer space to the BIOS if you answer the "Amount of space reserved in OS data segment" question in Figure 9-5 with a non-zero value.

Initialized data structures and buffers created by GENCPM become part of the CPM3.SYS file. Uninitialized areas are reserved for the operating system, but are not made part of the CPM3.SYS file. This keeps the CPM3.SYS file size to a minimum and improves system boot time. Appendix E contrasts the CPM3.SYS file and the memory image of CP/M-86 Plus.

GENCPM assumes the disk drivers are able to transfer data to and from memory wherever GENCPM places the directory and the data buffers.

If you want drives to share a linked list of buffers, define the number of buffers for one of the drives and use its drive letter for all other drives to share the buffers. Buffers are usually shared among drives to keep memory consumption down. Separate buffers can be useful though, when the physical sector sizes on different drives are highly disparate. In Figure 9-7, drives B: and C: share directory and data buffers with drive A:; drive E: shares only data buffers with drive D:. The buffer size of a shared list of buffers must be the largest sector size used by any of the drives.

Each drive must have at least one directory buffer available to it. Several drives can share a directory buffer, but if directory buffers are not shared, each drive must have at least one directory buffer of its own. Similarly if the sector size is larger than 128 bytes, each drive must have at least one data buffer available to it.

Question Variable: PARMDRVd where d = drives A - P.

Option 6: Generate a System and Exit

Selecting option 6 of the Main Menu creates a new CPM3.SYS file, and optionally, a new GENCPM.DAT file to be generated. Option 6 displays the following screen:

```
CP/M-86 Plus ROMing Information

                          Base   Length
                          ----   ------
System Code               0040H  052CH
Initialized System Data   056CH  018AH
Total System Data         056CH  0879H

Operating System Memory Table:
Partition  Base   Length
-------------------------
    0      0DE5H  011BH
    1      2001H  0FFFH

CPM3.SYS file created on drive B:

*** CP/M-86 Plus SYSTEM GENERATION DONE ***
```

Figure 9-8. GENCPM Generate a System and Exit Screen.

This information is of importance if you are placing CP/M-86 Plus in ROM, which is covered in Appendix G. The System Code is the segment address and legnth in paragraphs of the CP/M-86 Plus code segment. The Initialized System Data is the segment address and length in paragraphs of data that must be copied from ROM to the RAM data area specified by the answer to "Data Base of CP/M-86 Plus" question shown above in Option 3. The length in paragraphs in the Total System Data is the amount of contiguous RAM needed for the initialized and uninitialized data areas for the operating system. The memory table shows the memory partitions defined in Option 4 after they have been trimmed for overlap with the operating system.

Option 7: Exit without Generating a System

Selecting option 7 of the Main Menu returns you to the CP/M-86 prompt. GENCPM does not modify any existing CPM3.SYS or GENCPM.DAT files.

There is a 'PARMDRV' Question Variable for each possible drive, A:-P:. If a drive's DPH offset in the BIOS Kernel Data Header @BH_DPHTABLE is 0, GENCPM ignores the corresponding 'PARMDRV' Question Variable. The three value parts of a 'PARMDRV' Question Variable are also ignored if a 0FFFFh value is not found in the corresponding DPH_DIRBUF, DPH_DATBUF, and DPH_HSTBL fields of the DPH.

End of Section 9

Section 10

BIOS Debugging

This section suggests a method of debugging CP/M-86 Plus that requires CP/M-86 1.X to be running on the target machine. It is also helpful to have a remote console, which can serve as the CP/M-86 1.X system console. Hardware-dependent debugging techniques, such as a ROM monitor and an in-circuit emulator, can also be used, but are not described in this manual.

Appendix A outlines an example series of BIOS implementation steps designed to minimize debugging time. Whatever steps you use, it easier to debug a BIOS using all polled device drivers as a first cut, then add interrupt-driven devices one at a time.

The tick interrupt routine is usually the last to be implemented. Remember to replace any hardware timing loops with calls to ?DELAY in the BIOS Kernel after the tick interrupt is running.

The initial system can be run without a tick interrupt, but has no way of forcing CPU-bound tasks to dispatch. However, without the tick interrupt, console and disk drivers are much easier to debug. In fact, if problems are encountered after the tick interrupt has been enabled, it is often helpful to disable it again to simplify the environment. Accomplish this by changing the tick interrupt handler to execute an IRET instruction instead of jumping to INT_DISPATCH and disabling the tick routine's CALLFs to INT_SETFLAG.

For you to debug CP/M-86 Plus using CP/M-86 1.X, the CP/M-86 1.X console device must be separate from the console used by CP/M-86 Plus. Usually, a terminal is connected to a serial port and the console input, console output, and console status routines in the CP/M-86 1.X BIOS are modified to access the serial port hardware. In other words, the CP/M-86 1.X logical CON: device must be mapped to another console device that is not used by CP/M-86 Plus.

You may need to modify the CP/M-86 1.X BIOS memory segment table to reflect the following requirements. Values in the CP/M-86 1.X BIOS memory segment table must not overlap memory represented by CP/M-86 Plus Transient Program Area (TPA). However, the CP/M-86 1.X BIOS must have in its memory segment table the area of RAM that the CP/M-86 Plus system image is to occupy. Thus DDT-86 or SID-86 can be loaded under CP/M-86 1.X, and the CPM3.SYS file read by the debugger to the memory location you specify to GENCPM. The following figure illustrates one possible memory organization for debugging CP/M-86 Plus under CP/M-86 1.X.

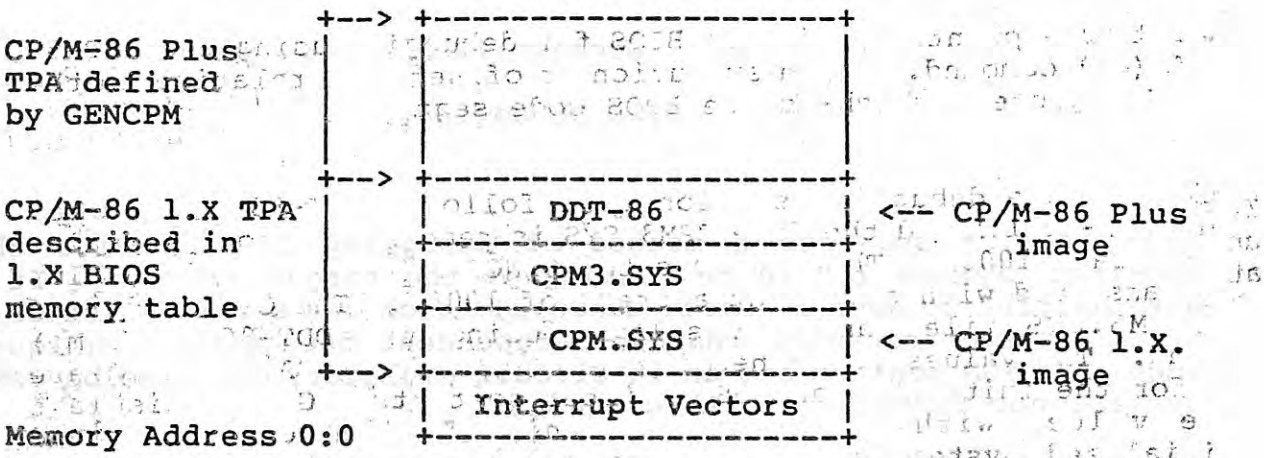


Figure 10-1. Debugging Memory Organization

Any hardware shared by both CP/M-86 1.X and CP/M-86 Plus systems is usually not accessible to CP/M-86 1.X after CP/M-86 Plus has completed its initialization. Typically, this prevents you from getting out of DDT-86 and back to CP/M-86 1.X, or executing any disk I/O under DDT-86. DDT-86 and SID-86 use interrupt vectors 1, 3, and 225, which must be preserved by the CP/M-86 Plus BIOS initialization routines. If CP/M-86 1.X uses any interrupt vectors for I/O to the remote console, these interrupt vectors must also be preserved by the CP/M-86 Plus BIOS initialization routines.

The following outline describes the technique for debugging the CP/M-86 Plus BIOS with DDT-86 running under CP/M-86 1.X:

1. Switch the CP/M-86 1.X logical CON: device to a remote console. This remote console cannot be accessed by the CP/M-86 Plus BIOS. (Some CP/M-86 1.X systems can accomplish this mapping through the IOBYTE and the 1.X STAT utility.)

2. Ensure the CP/M-86 1.X and CP/M-86 Plus TPAs are set as specified above.

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

4. Load the CPM3.SYS file under DDT-86 using the R (Read) command and the starting segment address of the CP/M-86 Plus system minus 8 (the length in paragraphs of the CMD file header). You specify this address with the "Base of CP/M-86 Plus ?" question in GENCPM. Set up the CS and DS registers from the A-BASE values found in the CPM3.SYS CMD file header record. See Appendix E in the Programmer's Guide for a description of the CMD file header.

5. The double word addresses for the BIOSENTRY and BIOSINIT routines are in the SYSDAT DATA at offsets 28h and 2Ch respectively.

6. Set breakpoints within the BIOS for debugging using the DDT-86 G (Go) command. Begin execution at offset 0 of relative to the CS register, which is the BDOS code segment.

In the example debugging session that follows, DDT-86 is invoked under CP/M-86 1.X and the file CPM3.SYS is read into memory starting at paragraph 1000h. This assumes the GENCPM "Base of CP/M-86 Plus ?" was answered with a segment address of 1008H. The CMD header of the CPM3.SYS file can be displayed using the DDT-86 D (Dump) command. The values contained in the CMD header A-BASE fields are used for the initial CS and DS register contents. GENCPM displays these values with the lines beginning "System Code ..." and "Initialized System Data ..." that are printed at the end of a GENCPM session.

Listing 10-1 shows two DDT-86 G (Go) commands with breakpoints, one to the beginning of the BIOSINIT routine and the other to the beginning of the BIOSENTRY routine. When these breakpoints are executed DDT-86 and CP/M-86 1.X regain control and print a prompt on the remote console. The CP/M-86 Plus BIOS routines can now be "single-stepped" using the DDT-86 T (Trace) command, or other breakpoints can be set within the BIOS. See the Programmer's Guide for more information on DDT-86. In Listing 10-1, the system implementor's responses are in bold type.

Listing 10-1. DDT-86 Example Debugging Session

```
A> ddt86
DDT86
-rcpm3.sys,1000:0
  START       END
1000:0000  1000:NNNN
-d0
1000:0000  01 12 06 08 10 12 06 00 00 02 B9 08 ED 14 B9 08 ........
-xcs
CS 0000 1008 <--------+
DS 0000 14ed <--------------------------------------+
SS 0051 .

-swl4ed:28
14ED:0028 0003          ;use the S (set) command here to
14ED:002A 1455          ;display but not set memory values
14ED:002C 0000
14ED:002E 1455
161A:0030 XXXX .
-g,1455:0               ;set a breakpoint at BIOS INIT
*1455:0000              ;the INIT routine may now be debugged
```

Listing 10-1. (continued)

```
-g,1455:3                ;set a breakpoint at BIOS ENTRY
*1455:0003               ;the BIOS function being called
      .                  ;is in AL.
      .
      .
```

If you are running DDT-86 the BIOS Kernel code offsets in memory are the same as the listing generated by RASM-86. The BIOS Kernel data offsets have 0F00h added to them since the BIOS data begins at 0F00h. To find code and data symbols in the other BIOS modules, use the RASM-86 $LO option, which causes LINK-86 to include local symbols into the symbol file. The symbol file can be printed out and used as a cross-reference when debugging with DDT-86. A map file of the BIOS modules generated by the LINK-86 $MAP option can also be helpful. See the Programmer's Utilities Guide for more information on RASM-86 and LINK-86.

SID-86 simplifies debugging because code and data can be accessed by symbol names. SID-86 reads the symbol file generated by LINK-86 to find the offsets of requested symbols. When using SID-86 extend the CPM3.SYS file to include uninitialized buffers not in the file. These buffers include the allocation, checksum, disk, and directory buffers. (See Appendix E.) This ensures the symbols are not placed in memory by SID-86 where they will be written over during the debugging session. Assuming the same CPM3.SYS file as in the previous example, Listing 10-2 illustrates how to extend the file. The system implementor's responses are in bold type.

Listing 10-2. SID-86 Example Debugging Session

```
A> sid86
#rcpm3.sys,1000:0
  START      END
1000:0000 1000:NNNN
#xcs
CS 0000 1008
DS 0000 14ed
SS 0051 .
#sw14ed:48
14ED:0048 xxxx .           ;OSENDSEG value from SYSDAT DATA
#wcpm3.sys,1000:0,XXXX:0   ;note the CPM3.SYS file needs only
      .                    ;to extended once after system
      .                    ;generation by GENCPM.
      .
#e                         ;release memory allocated to SID86
#rcpm3.sys,1000:0          ;read in larger file
  START      END
```

Listing 10-2. (continued)

```
1000:0000 -1000:ZZZZ
#e*bios3                ;now we can get the BIOS3.SYM file
SYMBOLS
#g,.BIOSINIT            ;a '.' requests a symbol lookup by SID-86
     .                  ;continue debugging as outlined above for
     .                  ;DDT86
```

End of Section 10