CPMPIG0.WS4 (= "CP/M-86 Plus Installation Guide", section 0)

CP/M-86 Plus -- Installation Guide

****
* First Edition: November 1983 *
****

(Edited by Emmanuel ROCHE.)

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 Installation Guide", (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 unfamiliar 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 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. The loader can be completed before the BIOS, since it entails 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 loader 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, DDT-86, and SID-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 User's Guide" (hereinafter cited as the "User's Guide") describes CP/M-86 Plus, and explains how to use its utilities and other features used to execute application programs.

- The "CP/M-86 Plus 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 synonymously 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, documented in the "Programmer's Guide", 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_".

- 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 hexadecimal (base 16) or a "b" denoting bits. However, the default base for DDT-86 and SID-86 is hexadecimal.

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

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

Table of Contents

Appendixes

Tables, Figures, and Listings

Tables

Figures

Listings

EOF

CPMPIG1.WS4 (= "CP/M-86 Plus Installation Guide", section 1)

(Edited by Emmanuel ROCHE.)

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:

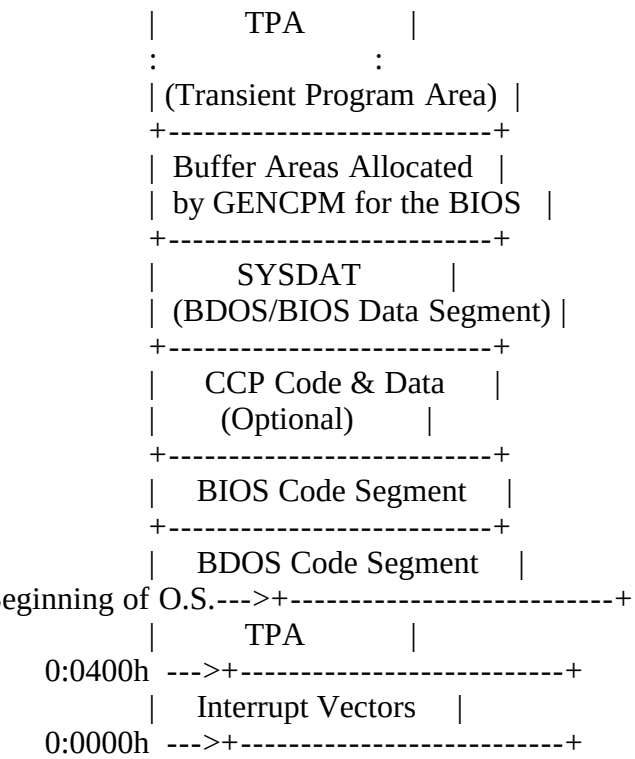


Figure 1-1. General Memory Organization of CP/M-86 Plus

The BDOS is the invariant and logical nucleus of CP/M-86 Plus. Six sub-modules divide the nucleus into the following functional groupings:

- Entry to and exit from 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 invariant 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. 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.

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.

MODULE COMMUNICATION

After the system initialization, the BDOS passes control to the CCP. If the CCP is not part of the operating system image, the BDOS loads it from disk and the CCP 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. The "Programmer's Guide" documents these system calls and they are all implemented within the BDOS.

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 Record, 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 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 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 read and write operations transfer physical sectors up to 16 Kbytes at a time.

- GENCPM creates the CP/M-86 Plus image contained in the CPMP.SYS file and provides many configuration options. GENCPM can automatically allocate all 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.

EOF

CPMPIG10.WS4 (= "CP/M-86 Plus Installation Guide", section 10)

(Edited by Emmanuel ROCHE.)

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.

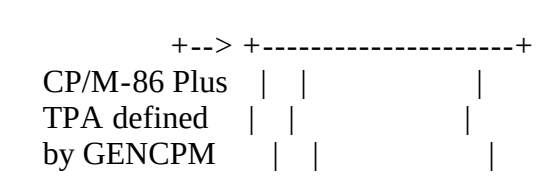
Appendix A outlines an example series of BIOS implementation steps designed to minimize debugging time. Whatever steps you use, it is easier to debug a BIOS using polled device drivers as a "first cut", then add interrupt-driven devices one at a time.

The tick interrupt routine is usually the last to be implemented. Remember to replace any software CPU delay 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 performing JMPF (Jump Far instruction) to INT_DISPATCH, and by 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 CPMP.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.



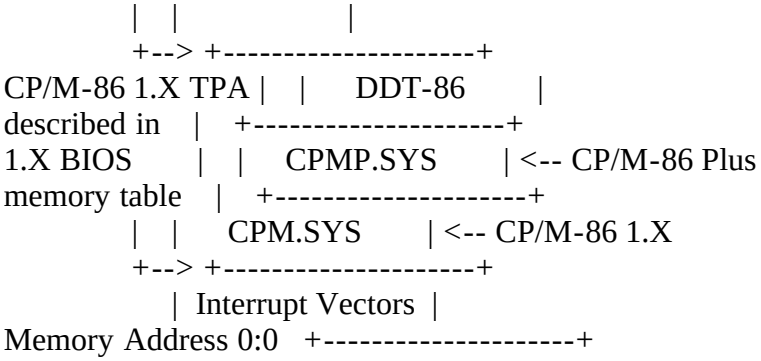


Figure 10-1. Debugging Memory Organization

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. Read the CPMP.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 Record). You specify the starting segment 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 CPMP.SYS CMD file Header Record. See Appendix D in the "Programmer's Guide" for a description of the CMD file Header Record.

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 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 CPMP.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 file Header Record of the CPMP.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.

Listing 10-1. DDT-86 Example Debugging Session

```
A>ddt86
DDT86
-rcpmp.sys,1000:0
  START      END
1000:0000  1000:NNNN
-d0
1000:0000  01 FE 04 08 10 FE 04 00 00 02 92 01 06 15 92 01 ......
               +--+--+               +--+--+
-xcs               |                   |
CS 0000 1008 <--------+                   |
DS 0000 1506 <-----------------------------------+
(...)
-sw1506:28
1506:0028 0003         ;use the S (Set) command here to
1506:002A 1467         ;display but not set memory values
1506:002C 0000
1506:002E 1467
1506:0030 XXXX .
-g,1467:0              ;set a breakpoint at BIOS INIT
*1467:0000               ;the INIT routine may now be debugged
(...)
-g,1467:3              ;set a breakpoint at BIOS ENTRY
*1467:0003               ;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 CPMP.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 that the symbols are not placed in memory by SID-86 where they will be written over during the debugging session. Assuming the same CPMP.SYS file as in the previous example, Listing 10-2 illustrates how to extend the file.

Listing 10-2. SID-86 Example Debugging Session

```
A>sid86
#rcpmp.sys,1000:0
  START     END
1000:0000 1000:NNNN
#xcs
CS 0000 1008
DS 0000 1506
(...)
#sw1506:48
1506:0048 XXXX .         ;OSENDSEG value from SYSDAT DATA
#wcpmp.sys,1000:0,XXXX:0  ;note the CPMP.SYS file needs to be
(...)                ;extended only once after system
(...)                ;generation by GENCPM.
#e                    ;release memory allocated to SID86
#rcpmp.sys,1000:0        ;read in larger file
  START      END
1000:0000  1000:ZZZZ
#e*bios3               ;now we can get the BIOS3.SYM file
SYMBOLS
#g,1467:.biosinit        ;a '.' requests a symbol lookup by SID-86
(...)                ;continue debugging as outlined above for
(...)                ;DDT-86
```

EOF

CPMPIG11.WS4 (= "CP/M-86 Plus Installation Guide", section 11)

(Edited by Emmanuel ROCHE.)

Section 11: System Boot Operations

Boot operations read the operating system image into memory, then transfer control to it. You can accomplish this in a variety of ways, including the use of an already existing MP/M-86 or Concurrent CP/M loader. This section presents the boot operations on the CompuPro 8/16 for you to use as a model in customizing your own hardware environment. Appendix G discusses loading CP/M-86 Plus in a ROM-based system.

BOOT OVERVIEW

This example CP/M-86 Plus implementation on the CompuPro involves a four-step procedure. First, a ROM in the CompuPro reads the disk boot loader from track 0 drive A:, then transfers control to it. Second, the disk boot loader reads the remaining drive A: boot tracks, which contain CPMLDR (the CP/M-86 Plus loader), then it passes control to CPMLDR. Third, CPMLDR reads the CPMP.SYS file into memory, then transfers control to the BDOS initialization entry point. The fourth step is system initialization which occurs in the BDOS and the BIOS.

In this fourth step, the BDOS executes its internal initialization routines, and calls the BIOSINIT entry point in the system BIOS. The BIOS initialization routines perform any remaining hardware initialization not done in the disk boot loader or in CPMLDR. The BIOS returns to the BDOS, and the BDOS passes control to the CCP, which prompts for commands from the user. Section 5 presents system and BIOS initialization in more detail.

Memory areas used by succeeding steps of the boot sequence cannot overlap. For example, the memory used by the disk boot loader must be distinct from the memory used by CPMLDR, and the memory CPMLDR occupies cannot overlap the target memory for the CPMP.SYS system image.

The distribution disks contain the files used to construct a CP/M-86 Plus disk boot loader and CPMLDR for the Compupro 8/16. These files are DSKBOOT.A86, the disk boot loader; LPROG.A86, the loader program; LBIOS.A86, the loader BIOS; and LBDOS3.SYS, the loader BDOS. The DSKBOOT.A86, LPROG.A86, and the LBIOS.A86 source files must be customized for your machine, assuming a similar load sequence as the one used for the CompuPro. The GENLDR utility creates CPMLDR using the LBDOS3.SYS file and your customized loader program and loader BIOS.

COMPUPRO TRACKS 0 AND 1

CP/M-86 Plus reserves tracks 0 and 1 on a CompuPro boot disk for different

parts of the boot operation. (See the DPH_OFF field in the "Disk Parameter Header" subsection in Section 7.) The rest of the tracks are reserved for directory and file data. Track 0 is divided into two areas: sectors 1-4, inclusive, contain the disk boot loader, and sectors 5-26 contain the first part of CPMLDR. It is assumed that track 0 on CompuPro CP/M-86 Plus boot disk is always in single-density format with 26 sectors each 128 bytes long. Track 1 contains the rest of CPMLDR. Figure 11-1 shows the layout of track 0 of a CP/M-86 Plus boot disk for the CompuPro 8/16.

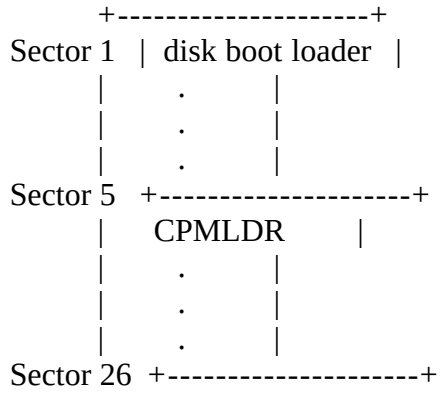


Figure 11-1. Track 0 on the CompuPro 8/16

Track 1 and the rest of the disk on the CompuPro can be one of several different formats. The example disk boot loader determines the format of track 1 before reading the rest of CPMLDR.

DISK BOOT LOADER

The disk boot loader loads CPMLDR from the boot tracks on the boot disk, and transfers control to it. The disk boot loader usually must be small, since space on the boot tracks is limited. The disk boot loader resides in a special area of the boot disk that is read by a ROM at power-on or reset.

The disk boot loader must read the first sector of the CPMLDR, and determine from the CMD file Header Record the memory location in which to place CPMLDR. This segment address is in the A-BASE field in the code group descriptor in the CPMLDR.CMD file's CMD Header Record, which is in the first 128 bytes of CPMLDR. (Appendix D in the "Programmer's Guide" shows the CMD file Header Record format.) Note that the GENLDR sets the G_LENGTH field in CPMLDR's code group descriptor to the length of CMPLDR code and data, not just the code. Once the disk boot loader has read CPMLDR to the requested memory location, the boot loader performs a JMPF (Jump Far instruction) to the CPMLDR code segment, offset 0. This code segment is also the CPMLDR code group's A-BASE value.

The example CompuPro disk boot reads sector 5 from track 0, which is the CPMLDR CMD file Header Record, to determine where to read CPMLDR. The disk boot loader then reads the rest of track 0, sectors 6-26, which contain the first part of CPMLDR. The disk boot loader determines the format of the rest of the disk, so track 1 and the last part of CPMLDR can be read next.

CP/M-86 PLUS LOADER: CPMLDR

CPMLDR is a simple version of CP/M-86 Plus that contains sufficient file processing capability to read the operating system image file, CPMP.SYS, from the boot disk to memory. When CPMLDR completes its operation, the operating system image receives control, and CP/M-86 Plus begins execution.

CPMLDR consists of three modules: the loader BDOS, the loader program, and the loader BIOS. The loader BDOS is used by the loader program to open and read the system image file from the boot disk. The loader program opens and reads the CPMP.SYS file, prints the loader sign-on message, and transfers control to the system image. The loader BIOS implements the hardware specific routines required by the loader BDOS and the loader program. The loader BIOS must be customized for your hardware, the loader program can be optionally changed, and the loader BDOS is invariant for any CP/M-86 Plus CPMLDR.

Figure 11-2 shows the layout of the loader BDOS, the loader program, and the loader BIOS within CPMLDR. The loader BDOS uses the three-entry jump table to pass control to the loader program and the loader BIOS. This jump table must follow immediately after the loader BDOS code, and is usually contained in the loader BIOS.

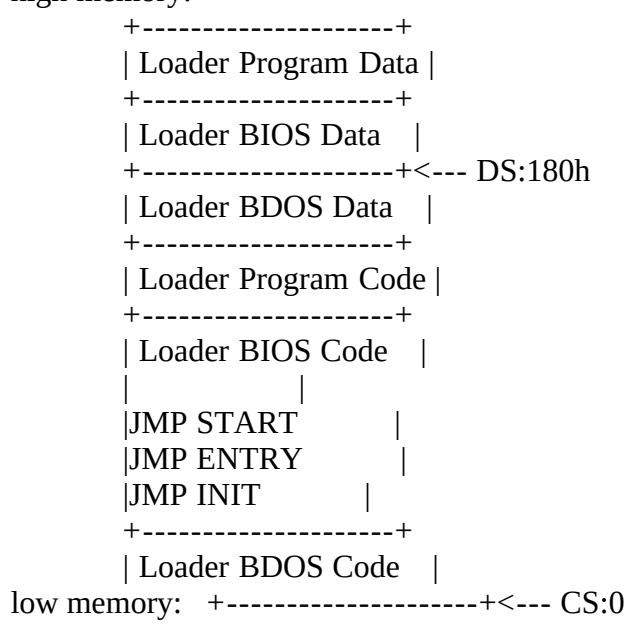


Figure 11-2. CPMLDR Organization

After receiving control from the disk boot loader, the loader BDOS sets the DS and ES registers to the CPMLDR data segment. The SS and SP registers are set to a 64-level stack (128 bytes) within the loader BDOS. The assumption is that the loader BDOS, the loader program, and the loader BIOS will not require more than 64 levels of stack. If this is not true, then the loader program, the loader BIOS, or both, must perform a stack switch when necessary. The three CPMLDR modules (the loader BDOS, program, and BIOS) execute using a separate code and data model.

The GENLDR utility creates CPMLDR using the files LBDOS3.SYS and LBIOS3.SYS as input. The LBIOS3.SYS file you create must contain the loader BIOS Data Header and the loader BIOS Code Header. GENLDR requires these headers. The loader BIOS Code Header consists of three JMP instructions and one word value, making it 11 bytes long. Listing 11-1 shows the loader BIOS Code Header from the LBIOS.A86 on the distribution disks.

Listing 11-1. Loader BIOS Code Header

```
        CSEG

        public  @sysdat
        public  ?conout, ?pmsg
        extrn   ?start:near

        ORG     0000h

        jmp init                ;I/O system initialization
        jmp entry               ;I/O system function entry
        jmp ?start              ;start of loader program

@sysdat         rw      1               ;CPMLDR data segment
                                ;forces CS override
```

The START label is the beginning of the loader program, and is declared a public symbol in the LPROG.A86 file. The INIT and ENTRY labels address the loader BIOS initialization routine and the loader BIOS function entry point. @SYSDAT is the data segment of CPMLDR that is also the data segment for the loader BIOS and the loader program. Data for the loader program and the loader BIOS must start at 180h; the prior bytes in the CPMLDR data segment are reserved for loader BDOS. Use the LINK-86 ORIGIN option to place the loader program and loader BIOS data at 180h. (See "Boot Tracks Construction" later in this section.)

Listing 11-2 shows the loader BIOS Data Header, which consists simply of one word that must be at location 180h in the CPMLDR data segment. This is the offset of the Disk Parameter Header for the boot drive. GENLDR locates the DPH, and can optionally create one directory buffer and one data buffer for the loader BIOS.

Listing 11-2. Loader BIOS Data Header

```
        DSEG

        public  @bh_dphtable, @dpha

        ORG     0000h
                ;use the LINK-86 [data[origin[0180]]] option
                ;to set the origin of the data segment at 0180h

;       disk parameter header offset table

@bh_dphtable    dw      offset @dpha    ;drive A:
```

After initializing local variables and the segment registers, the loader BDOS performs a CALLF (Call Far instruction) to JMP INIT in the loader BIOS Code Header. The INIT routine is the loader BIOS initialization routine, and must return to the loader BDOS using RETF (Return Far instruction). The loader BDOS initializes interrupt vector 224, then performs a JMPF (Jump Far instruction) to JMP START in the loader BIOS Code Header. The START label is the beginning of the loader program. The START label is declared as a public symbol in the example loader program, and as an external in the example loader BIOS.

The loader program performs INT 224 instructions for functions supported in the loader BDOS. The loader BDOS performs a CALLF (Call Far instruction) to JMP ENTRY in the loader Code Header to invoke any required loader BIOS functions. The loader BIOS ENTRY routine returns to the loader BDOS by executing a RETF (Return Far instruction).

The loader BDOS does not turn interrupts on or off. If they are needed by the loader, they must be turned on by the ROM, the disk boot loader, or the loader BIOS. The example loader BIOS executes a STI (Set Interrupt Enable Flag instruction) in the loader BIOS INIT routine.

Typically, the loader program prints out a short sign-on message by calling a routine in the loader BIOS directly. The loader program then opens and reads the CPMP.SYS file using the CP/M-86 Plus system calls supported by the loader BDOS. The loader program sets the DS register to the A-BASE value found in the CPMP.SYS CMD file Header Record's data group descriptor. (Appendix D in the "Programmer's Guide" describes the CMD file Header Record format.) Finally, the loader program transfers control to CP/M-86 Plus with a JMPF (Jump Far instruction) at the end the loader program.

LOADER BDOS AND LOADER BIOS FUNCTION SETS

The loader BDOS implements the minimum set of system calls required to open the CPMP.SYS file, and to transfer it to memory. Invoke these system calls as under CP/M-86 Plus by executing an INT 224 (00E0H). (The "Programmer's Guide" documents system calls.) The necessary system calls are in the following table. Any other function, if called, returns a 0FFFFh error code in registers AX and BX.

Table 11-1. Loader BDOS System Calls

Number	Mnemonic	Description
14 0Eh	DRV_SET	Select Disk
15 0Fh	F_OPEN	Open File
20 14h	F_READ	Read Sequential
26 1Ah	F_DMAOFF	Set DMA Offset
32 20h	F_USERNUM	Set/Get User Number
44 2Ch	F_MULTISEC	Set Multisector Count
51 33h	F_DMASEG	Set DMA Segment

Blocking/Deblocking has been implemented in the loader BDOS, as well as Multisector disk I/O.

The loader BIOS must implement the following functions required by the loader BDOS to read a file.

Table 11-2. Required Loader BIOS Functions

Number	Mnemonic	Description
9 09h	IO_SELDSK	Select Disk
10 0Ah	IO_READ	Read Physical Sectors

The loader BIOS functions are implemented in the same way as the corresponding CP/M-86 Plus operating system BIOS functions. Therefore, the code used for the loader BIOS can be a subset of the system BIOS code. The loader BIOS, however, cannot use the ?FLAGWAIT, ?DISPATCH, ?DELAY, or INT_FLAGSET functions, which are available in the system BIOS. When waiting for hardware events in the loader BIOS, test the hardware status within software loops.

Since the loader BDOS performs only read operations, certain disk data structures are not needed. The DPH_CSV, DPH_ALV, DPH_HSHTBL, DPH_WRITE, and DPH_FLAGS fields in the Disk Parameter Header are not used by the loader BDOS or GENLDR. Similarly, the Disk Parameter Block (DPB) fields DPB_AL0, DPB_AL1, and DPB_CKS are unused by the loader BDOS or GENLDR.

The loader BIOS must have two physical sector disk buffers, one for the directory, and another for data. Any disk buffers past these two are not used by the loader BDOS. If the DPH_DIRBCB is defined in the loader BIOS equal to 0FFFFh, GENLDR allocates one BCB Header, DIRBCB, and directory buffer. Similarly, if DPH_DATBCB is 0FFFFh, GENLDR allocates one BCB Header, DATBCB, and data buffer. The data and directory buffers are not made part of the CPMLDR.SYS file generated by GENLDR, but are reflected in the G-LENGTH field in the code group descriptor in the CPMLDR.SYS CMD file Header Record.

Note: If you define the data buffer data structures in the loader BIOS, the loader BIOS INIT routine must change the BCB_BUFSEG address in the DATBCB to be the segment address of the data buffer. "DPH_HSHTBL and BCB_BUFSEG Initialization" in Section 7 covers this topic for the system BIOS, and is also applicable here.

The maximum size of the loader BIOS and the loader program depend on the space left on the boot tracks after the disk boot loader. To keep the CPMLDR size small, the loader BIOS and the loader program are contiguous. The example loader BIOS implements two public console output routines, ?CONOUT and ?PMSG. If desired, the loader BIOS can be expanded to support keyboard input, to allow the loader program to prompt for user options at boot time. However, the only loader BIOS functions invoked by the loader BDOS are IO_SELDSK and IO_READ. Any other loader BIOS functions must be invoked directly by the loader program. The example loader program (LPROG.A86) calls ?CONOUT and ?PMSG in the loader BIOS.

BOOT TRACKS CONSTRUCTION

Use the following procedure to create the loader from the LBDOS3.SYS file and the DSKBOOT.A86, LBIOS.A86, and LPROG.A86 source files.

First, use the following commands to create an executable disk boot loader file:

```
A>RASM86 DSKBOOT
A>LINK86 DSKBOOT.SYS=DSKBOOT [DATA[ORIGIN[0]]]
```

Note that the file created cannot be used as a transient, nor can the next file you create.

Next, create the executable CPMLDR.SYS file. First, assemble the loader BIOS and the loader program contained in the LBIOS.A86 and LPROG.A86 files. Link these files together to create the file LBIOS3.SYS, as shown in the next example. The LINK-86 ORIGIN option must be used to set the origin of the LBIOS3.SYS data to 180h. Your file that contains the loader BIOS data and code headers must be the first OBJ file specified on LINK-86 command tail. For the example loader, this is the LBIOS.OBJ file, as shown here:

```
A>RASM86 LBIOS
A>RASM86 LPROG
A>LINK86 LBIOS3.SYS=LBIOS,LPROG [DATA[ORIGIN[180]]]
```

Then, use GENLDR to create CPMLDR.SYS from LBDOS3.SYS and the LBIOS3.SYS files.

```
A>GENLDR [1008]
```

The option in this last command specifies the segment address where CPMLDR is placed and executed in memory. This example command results in CPMLDR being placed at location 1008:0000h. GENLDR places this address value in three places in the CPMLDR.SYS file. The first location is in the @SYSDAT word in the loader code header shown in the example. The second is within the loader BDOS code segment at offset 6, which the loader BDOS uses to set DS and ES after receiving control from the disk boot loader. The third is in the A-BASE field of the CPMLDR.SYS file's CMD Header Record code group descriptor. As mentioned earlier, GENLDR optionally allocates disk I/O buffers, one each for directory and data.

Next, combine the DSKBOOT.SYS and CPMLDR.SYS files into a track image using one of the debuggers DDT86 or SID-86, and the PIP utility. The following example names the track image file BOOTTRKS. (The PIP O (Object) option used in the example specifies object file concatenation.)

Listing 11-3. Boot Tracks Construction

```
A>SID86
#rdskboot.sys          ;load in the DSKBOOT.CMD file
  START      END       ;aaaa is the segment address
aaaa:0000 aaaa:37F     ;paragraph where SID86 places BOOT.CMD
#wboot,180,37f         ;create the file BOOT
#^C                    ;without a CMD file Header Record
A>PIP BOOTTRKS=BOOT[O],CPMLDR.SYS[O]
```

As a final step, place the contents of the BOOTTRKS file onto tracks 0 and 1. Use the TCOPY program to accomplish this:

```
A>TCOPY BOOTTRKS  ;copy the BOOTTRKS file to tracks 0 and 1
```

You must run TCOPY under a CP/M-86 1.X system, since it makes direct BIOS calls supported only under CP/M-86 1.X. You must modify TCOPY for different disk formats, and for a different number of tracks to write. The source is included in the file TCOPY.A86 on the distribution disks.

DISK BOOT AND CPMLDR DEBUGGING

When debugging a customized disk boot loader, CPMLDR, or TCOPY program, use scratch diskettes.

Use DDT-86 or SID-86 to debug CPMLDR, which can be done separately from debugging the disk boot loader. Listing 11-4 shows an example of using SID-86 to debug the CPMLDR:

Listing 11-4. Sample Debugging of the CPMLDR

```
A>SID86
#rcpmldr.sys,1000:0
  START     END        ;aaaa is the segment location;
aaaa:0000 aaaa:xxxx     ;xxxx is the size of the CompuPro
                  ;CPMLDR used for this example.
#d0                 ;show CPMLDR CMD file Header Record
1000:0000  01 LL LL CC CC 00 00 00 00 00 00 00 DD DD 00 00 .....
#xcs
CS 0000 CCCC
DS 0000 DDDD
(...)
#dwds:0
DDDD:0000  0003 XXXX 0000 XXXX 0006 XXXX ???? ???? .............

        (ENTRY)   (INIT)    (START)
#g,XXXX:3             ;breakpoint at loader BIOS
*XXXX:0003             ;initialization
(...)
#g,XXXX:0             ;breakpoint at loader BIOS
*XXXX:0000             ;entry; the BIOS function
                  ;being called is in AL
(...)
#g,XXXX:6             ;breakpoint at beginning
*XXXX:0006             ;of loader program
(...)
```

This debugging technique requires that the memory for CPMLDR be available to DDT-86 or SID-86. Use GENLDR segment address parameter to change where CPMLDR loads. A remote console (see Section 10) is needed if the loader BIOS reinitializes the system console hardware. The debuggers use interrupt 225 to

communicate with the operating system. This does not conflict with interrupt 224 used by CPMLDR. However, you cannot exit back to the operating system after CPMLDR has changed interrupt vector 224, unless you copy vector 225 to vector 224 before typing Ctrl-C at the debugger prompt.

The addresses of the JMP table located in the loader data header are found as shown in the above SID-86 session. The addresses of the JMP INIT, JMP ENTRY, and JMP START instructions are double word pointers in the CPMLDR data segment, as shown. Using these double word addresses, breakpoints can be set at the loader BIOS initialization, the loader BIOS entry, and the beginning of the loader program.

To find locations of symbols when using SID-86, use the RASM-86 $LO option when assembling the loader program and the loader BIOS. This causes LINK-86 to generate the symbol file LBIOS3.SYM, which contains the symbols local to the loader program and the loader BIOS. Print out the symbol file, and use it as a cross-reference while debugging.

SID-86 simplifies debugging because the SYM file created by LINK-86 when the LBIOS3.SYS file is generated can be used directly with SID-86 to locate symbols in the CPMLDR image.

To test CPMLDR, use it to load a dummy CPMP.SYS file that simply prints a message to the screen to announce its presence.

Use DDT-86 or SID-86 to debug the disk boot loader in the same way you debugged CPMLDR.

OTHER BOOT METHODS

Many departures from this example disk boot operation are possible, and they depend on the hardware environment and your goals. For instance, the boot loader can be eliminated if the system ROM (or PROM) can read in the entire CPMLDR at reset. Also, the CPMLDR can be eliminated if the CPMP.SYS file is placed on boot tracks and the ROM can read in these boot tracks at reset. However, the latter usually requires too many boot tracks to be practical. Alternatively, CPMLDR can be placed into a PROM and copied to RAM at reset, eliminating the need for any boot tracks. (Appendix G discusses placing CPMP.SYS in ROM.) Any initialization usually performed by the two modules must be performed in the BIOS initialization routines.

EOF

CPMPIG12.WS4 (= "CP/M-86 Plus Installation Guide", section 12)

(Edited by Emmanuel ROCHE.)

Section 12: Hardware-dependent Utilities

A CP/M-86 Plus implementation often requires OEM-supported utilities. These utilities are generally specific to the hardware, and usually include methods for formatting disks, and for copying one disk to another.

DIRECT BIOS OR HARDWARE ACCESS

When special OEM utilities bypass the BDOS by making S_BIOS (direct BIOS) system calls, or by going directly to the hardware, several programming precautions are necessary to prevent conflicts due to the CP/M-86 Plus multitasking environment. Take the following steps to prevent other processes from accessing the disk system:

1. Check for CP/M-86 Plus by using the S_BDOSVERS system call. S_BDOSVERS returns AX=1031 for CP/M-86 Plus. If the operating system is not CP/M-86 Plus, have the program print an error message and terminate.

2. Make sure there are no RSXs in memory. ANYRSX is an RSX that checks for other RSXs in memory. ANYRSX can be made part of your hardware-dependent program using the GENRSX utility. GENRSX is documented in Section 8 of the "Programmer's Guide". If there are RSXs, have the program print an error message and terminate.

3. Check for other concurrently running processes. Use the S_SYSVAR function to return the number of running process, which must be one. If there is more than one process, have the program print an error message and terminate. This prevents a background program from accessing the hardware, especially the disk drives, through the BDOS. This also prevents the hardware-dependent program from being run in the background via the BACK utility. (The "User's Guide" describes the BACK utility.)

4. You can now safely call BIOS functions, or access the hardware directly.

5. When operations are complete, and if your program accesses the disk hardware, force the disk system to be reset by making a DRV_ALLRESET system call. Resetting the drives ensures removable media drives are logged back in when next accessed. Your program can now terminate.

A example COPYDISK utility for the CompuPro 8/16 is found in source form on the distribution disk as the file COPYDISK.A86. The following subroutine from COPYDISK.A86 illustrates the preceding steps for a hardware-dependent utility. The CHK_ENVIRONMENT routine, or one similar to it, must be called before

making any hardware-dependent operations.

Listing 12-1. Example for Hardware-dependent Utility

```
S_BDOSVER       equ     12          ;system call equates
S_SYSVAR        equ     49
P_RSX           equ     60

CPM_VERS        equ     1031h       ;CP/M-86 Plus version number
ANYRSXF         equ     150         ;ANYRSX subfunction number
NPROCS          equ     89h         ;S_SYSVAR subfunction number

        CSEG

chk_environment:
;---------------
        mov cl,S_BDOSVER            ;check version number
        int 224                     ;must be CP/M-86 Plus
        cmp ax,CPM_VERS
        jne errout

        mov dx,offset rsx_pb        ;call ANYRSX, if other RSXs
        mov cl,P_RSX                ;in memory byte 2 in the RSX
        int 224                     ;parameter block is set to 0
        test byte ptr rsx_pb+2,0FFh
        jnz errout

        mov bx,offset scb_pb        ;get the number running
        mov byte ptr [bx],NPROCS    ;processes from the BDOS
        mov byte ptr 1[bx],0
        mov dx,bx
        mov cl,S_SYSVAR
        int 224
        cmp scb_pb+2,1              ;can only have one running
        je envr_ok                  ;process
errout:
         mov si,offset errtoomany   ;print error message
         call pmsg
         jmp exit                   ;exit program
envr_ok:
        ret

        DSEG

rsx_pb          db   ANYRSXF,0,0FFh     ;RSX parameter block
scb_pb          db   0,0,0              ;sysdat parameter block

err_too_many    db   CR,LF,'This program cannot run when '
                db   'another program is running', CR,LF
                db   'in the background or when RSXs are present.'
                db   CR,LF,0
```

The source to ANYRSX is included on the distribution disk, in the file ANYRSX.A86. The commands to generate COPYDISK.CMD with the attached ANYRSX are

shown in the beginning comments of the COPYDISK.A86 source file.

DISK FORMATTING

A format utility, which you create and generally package with CP/M-86 Plus as a system utility, initializes fresh disk media for use with CP/M-86 Plus. The physical format of a disk is hardware-dependent, and therefore is not discussed here. This subsection discusses initialization of the directory area of a new disk.

A format program can initialize the directory with or without time and date stamping enabled. This can be a user option in the format program. If time and date stamps are not initialized, the user can independently enable this feature through the INITDIR and SET utilities, as documented in the "User's Guide".

It is highly recommended that you support the new features of CP/M-86 Plus by including time and date stamping in the format program. This simplifies the use of time and date stamping; otherwise, the user must first learn that date stamps are possible, then use the INITDIR and SET utilities to allow the use of this feature. Complications can arise with INITDIR if the disk directory is close to being full; INITDIR does not allow the restructuring of the directory that is necessary to include SFCBs. (The "Programmer's Guide" discusses SFCBs.)

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

A disk under CP/M-86 Plus is divided into three areas: the boot tracks, the directory area, and the data area. The Disk Parameter Block (see Section 7) determines the size of the directory and reserved areas. The directory area starts on the first disk allocation block boundary; the data area immediately follows the directory area. See Figure 7-1.

The boot tracks and the data area do not need to be initialized to any particular values before they serve as a non-bootable CP/M-86 Plus disk. The directory area, on the other hand, must be initialized to indicate that no files are on the disk. Also, as discussed later in this section, a format program can reserve space for time and date information, and can initialize the disk to enable this feature.

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

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

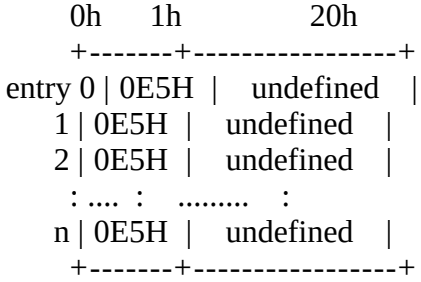


Figure 12-1. Directory Initialization Without Time Stamps

A disk directory initialized to enable time and date stamps must have SFCBs as every fourth Directory Entry. An SFCB has a 021H in the first byte, and all other bytes must be 0H. Also, a directory label must be included in the directory. This is usually the first Directory Entry on the disk. The directory label must be initialized as shown in Figure 12-2.

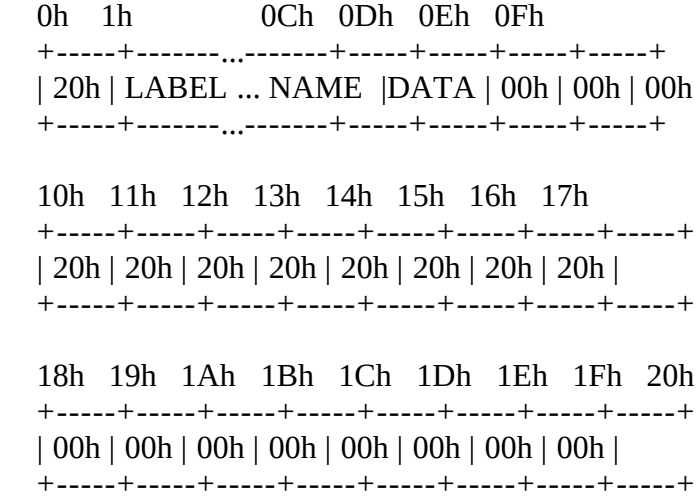


Figure 12-2. Directory Label Initialization

Table 12-1. Directory Label Data Fields

Format: Data Field
Explanation

LABEL NAME
An 11-byte field containing an ASCII name for the drive. The field follows the format for filenames: the first eight bytes are filename, and the last three bytes are the filetype. The filename and filetype parts of the label name should be blank-padded and left-justified.

DATA
A bit field that tells the BDOS general characteristics of files on the disk. The DATA field can assume the following values:

060H enables date of last modification, and date of last access to be updated, when appropriate.

030H enables date of last modification, and date of creation to be updated, when appropriate.

A format program should ask the user for the name of the disk, if time stamping is desired, and whether to use the date of last access or the date of creation, since these time stamps share the same field in the SFCB. The date of last modification has its own field in the SFCB, and can always be used. If the DATA field is 00H, or if the Directory Label does not exist, the time and date stamping is not enabled. The DATA Field must be 00H if SFCBs are not initialized in the directory.

Figure 12-3 shows a directory initialized for time and date stamping.

```
          0h     1h                        20h
          +------+-----------------------------+
   entry 0 | 020H | NAME,DATA  (Directory Label) |
         1 | 0E5H | undefined  (Unused)          |
         2 | 0E5H | undefined  (Unused)          |
         3 | 021H | NULLS      (SFCB)            |
         4 | 0E5H | undefined  (Unused)          |
         5 | 0E5H | undefined  (Unused)          |
         6 | 0E5H | undefined  (Unused)          |
         7 | 021H | NULLS      (SFCB)            |
          : .... : .........  ........          :
          | 0E5H | undefined  (Unused)          |
          | 0E5H | undefined  (Unused)          |
          | 0E5H | undefined  (Unused)          |
         n | 021H | NULLS      (SFCB)            |
          +------+-----------------------------+
```

Figure 12-3. Directory Initialization With Time Stamps

EOF

CPMPIG2.WS4 (= "CP/M-86 Plus Installation Guide", section 2)

(Edited by Emmanuel ROCHE.)

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 hardware-dependent modules you supply and the Kernel 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. It also 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)

All BIOS modules are separate code and data with any other segments, such as stack and extra segments, contained in the data segment.

Examples of these modules are on the distribution disks. These example modules implement a BIOS for operation on a CompuPro 8/16 with at least 64 Kbytes of RAM. However, 128 Kbytes of RAM or more is recommended. 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 disks in the file BIOSKRNL.A86.

All the BIOS Kernel functions, data structures, fields, public variables, and public subroutines 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 implementor.

- Debug the BIOS under CP/M-86 1.X using a remote console.

- Prepare, test, and debug the CP/M-86 Plus disk boot loader using an existing CP/M-86 1.X disk boot loader as a base.

- Prepare the CP/M-86 Plus loader program and BIOS, integrating with the loader BDOS to create the CP/M-86 Plus loader, CPMLDR. CPMLDR is built using the GENLDR utility.

- Use the TCOPY program to write the disk boot loader and CPMLDR onto the boot tracks of a boot disk.

- 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.

EOF

CPMPIG3.WS4 (= "CP/M-86 Plus Installation Guide", section 3)

(Edited by Emmanuel ROCHE.)

Section 3: BIOS Kernel

This section describes how the BIOS Kernel interfaces with the BIOS modules you supply and 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 address. 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, it is possible to access hardware-dependent information independent of a particular BIOS implementation. 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    ;auxiliary input
@bh_aoroot      dw      offset @cdbb    ;auxiliary 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
;       O.S. messages

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 command string
bh_cpmerr       dw      cmperr_msg  ;CP/M error message address
bh_func         dw      func_msg    ;function message address
bh_file         dw      file_msg    ;file message address
bh_err1         dw      err1_msg
bh_err2         dw      err2_msg
bh_err3         dw      err3_msg
bh_err4         dw      err4_msg
bh_err5         dw      err5_msg
bh_err6         dw      err6_msg
bh_err7         dw      err7_msg
```

The following table 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

Format: 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.

@BH_TICKSEC (0F01h)
[Number of system ticks per second]
This field is initialized by GENCPM but can be modified by the ?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 and does not 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_DOPEN in the Disk Parameter Header (DPH) associated with the drive. The DPH_DOPEN 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.

@BH_NEXTFLAG (0F04h)
This is the next system flag available for allocation to an interrupt routine. (Section 4 describes system flags.) The INIT routines for 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 0FFh, 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, Ctrl-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, which is explained later 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. When the DEVICE utility changes the @BH_CIROOT, it also updates @BH_INTCONIN.

@BH_8087 (0F07h)
This field is set to 0FFh by your INIT module if the 8087 is present and to 00h if it is not. Transient programs are marked as 8087 users by a field in the CMD file Header Record. 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. (RSXs cannot "own" the 8087.)

@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.)

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)
[Console Input Root]
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. Initialize this field with the CDB symbol (one of @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)
[Console Output Root]
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. Initialize this field with the CDB symbol (one of @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.

@BH_AIROOT (0F4Ch)
[Auxiliary Input Root]
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. Initialize this field with the CDB symbol (one of @CDBA-@CDBP) of the initial AUXIN: device. This CDB external is resolved by LINK-86 and can be 0.

@BH_AOROOT (0F4Eh)
[Auxiliary Output Root]
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. Initialize this field with the CDB symbol (one of @CDBA-@CDBP) of the initial AUXOUT: device. This CDB external is resolved by LINK-86 and can be 0.

@BH_LOROOT (0F50h)
[List Output Root]
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. Initialize this field with the CDB symbol (one of @CDBA-@CDBP) of the initial LST: device. This CDB external is resolved by LINK-86 and can be 0.

@BH_BUFBASE (0F52h)
This is the offset of the uninitialized buffer in the SYSDAT segment for use by the BIOS. Define the size and use of this area of RAM. The BDOS does not use this buffer. 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.

@BH_MEMDESC (0F56h)
[Memory Descriptor Table]
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 F 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 BDOS has released its memory. The offset in BH_CHAIN must address a printable string terminated by a "$". The default string defined in the BIOS Kernel is as follows:

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

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 "User's Guide" describes the Error CCP.) BH_PROMPT must address a printable string terminated by a "$". The default string defined in the BIOS Kernel is the following:

```
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)
The Error CCP uses the string addresses by this offset to recognize the one internal Error CCP command that changes user numbers. BH_USER must address a byte followed by the uppercase command. The first byte is the number of characters in the following string. The default string defined in the BIOS Kernel is as follows:

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

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

BH_CPMERR (101Ch)
BH_FUNC (101Eh)
BH_FILE (1020h)
The BDOS uses these three offsets to address strings for printing file-related error messages. The corresponding default strings as defined in the BIOS Kernel are the following:

```
cmperr_msg  db  13,10,'CP/M Error On $'
func_msg    db  13,10,'BDOS Function = $'
file_msg    db  ' File = $'
```

These strings can be changed to a foreign language as required. See Appendix H.

BH_ERR1 (1022h)
BH_ERR2 (1024h)
BH_ERR3 (1026h)
BH_ERR4 (1028h)
BH_ERR5 (102Ah)
BH_ERR6 (102Ch)
BH_ERR7 (102Eh)
The BDOS uses the strings addressed by these seven offsets to display a particular type of BDOS error. The corresponding default definitions in the BIOS Kernel are shown here:

```
err1_msg  db  'Disk Read/Write Error$'
err2_msg  db  'Read-Only Disk$'
err3_msg  db  'Read-Only File$'
err4_msg  db  'Invalid Drive$'
err5_msg  db  'Password Error$'
err6_msg  db  'File Exists$'
err7_msg  db  '? in Filename$'
```

These strings 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, as well as to 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 address, which is also the BIOS data segment, is kept in the code segment of the BIOS to be accessible from interrupt service routines.

The following code fragment from the BIOS Kernel shows the Code Header.

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 and the rest of 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 perform a 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
;
        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
```

As already mentioned, the BIOS Kernel assumes the other BIOS modules preserve DS and ES. 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 segment of currently running process environment is kept in the word 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_". The offsets of these functions are defined at the "FUNCTAB" symbol in Listing 3-3. The following table shows the two groupings of BIOS functions available to the BDOS:

Table 3-2. 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
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 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 Plus character devices CONIN:, CONOUT:, AUXIN:, AUXOUT:, and LST: to the physical character devices; they 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 3-3. 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 3-4 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 3-4. BIOS Kernel Character IO_ Functions

Format: Function
Definition

IO_CONIN
Calls the CDB_INPUT routine for the CDB addressed by @BH_CIROOT. If @BH_CIROOT is 0, then IO_CONIN 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 IO_AUXIN 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 IO_CONIST returns a 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 IO_AUXIST returns a 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 IO_CONOUT returns.

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 IO_AUXOUT 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 IO_LIST returns.

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 IO_CONOST also returns a ready status.

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 IO_AUXOST also returns 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 IO_LISTST also returns 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. DEVICE specifies the CDB offset as part of the S_BIOS call and the BDOS sets BX with this offset before calling IO_DEVINIT. IO_DEVINIT sets register DL to 1 before calling the CDB_INIT routine, indicating 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, which returns to the BDOS, and finally back to utilities such as DEVICE.

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 the CDB structure.

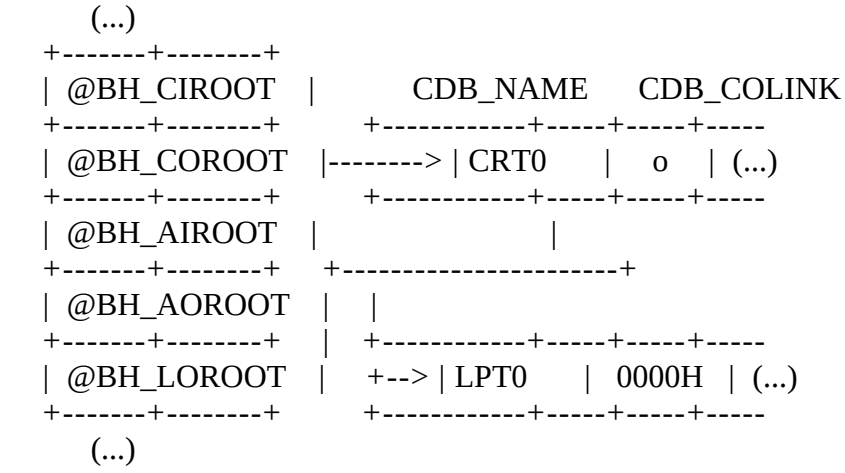


Figure 3-1. Character I/O Redirection Example

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:

Table 3-5 defines the BIOS Kernel Disk IO_ functions:

Table 3-5. BIOS Kernel Disk IO_ Functions

Format: 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 and 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. 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.

The DEVICE utility does not put the same CDB offset in both of the device input roots, @BH_CIROOT and @BH_AIROOT. Similarly DEVICE does not put the same CDB offset in more than one of the output character redirection lists rooted at @BH_COROOT, @BH_AOROOT, and @BH_LOROOT. Thus, when two or more processes perform I/O to the logical devices CON:, AUX: and LST:, the CDB routines in your CHARIO module are not reentered. Furthermore, the BDOS ensures two or more processes cannot access any one of the logical devices CON:, AUX:, and

LST:, simultaneously.

A BIOS routine that needs to make system calls back to the BDOS can do so by an INT 224 instruction, as described in the "Programmer's Guide", or by a CALLF (Call Far instruction) to the BDOS double word address in the SYSDAT segment (see Appendix C). Different system calls require differing amounts of stack; you may need to provide more stack for system calls made from the BIOS. The BDOS entry using the CALLF requires less stack and is more efficient than the INT 224 entry. The BIOS Kernel routines ?DISPATCH, ?DELAY, and ?WAITFLAG reach the BDOS with the CALLF BDOS instruction, and the register conventions for these routines are shown in the BIOS Kernel listing (see Appendix B). Note that, whenever the BDOS is called through the BDOS double word pointer in SYSDAT, the DS and ES registers must be set to the SYSDAT segment and the segment of the currently running process environment, respectively. The rest of the registers follow the conventions for a system call invoked via an INT 224 instruction.

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 3-6 shows the public BIOS Kernel routines that can be used by other BIOS Modules. Appendix B shows the register conventions for these routines.

Table 3-6. Public BIOS Kernel Routines

Format: 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 an INT_SETFLAG operation from a specific interrupt service routine. A process that must wait for an interrupt to signal the occurrence of a hardware event calls this routine. Each different interrupt-driven hardware event uses a different system flag number. You supply the flag number associated with a specific event as a parameter to ?WAITFLAG and INT_SETFLAG. System flags are allocated using the @BH_NEXTFLAG and @BH_LASTFLAG fields in the Kernel Data Header. Section 4 further discusses system flags and interrupt service routines.

?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.

EOF

CPMPIG4.WS4 (= "CP/M-86 Plus Installation Guide", section 4)

(Edited by Emmanuel ROCHE.)

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, CP/M-86 Plus supports a BIOS using polled I/O drivers with no interrupts and no tick.

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 stopping (Ctrl-S) and starting (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 output sent to the CON: device (usually the console) on the LST: device (usually 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 occurrence of the hardware event by performing a CALLF (Call Far instruction) to the BDOS INT_SETFLAG function with the same flag number.

The system flags are data structures manipulated by the ?WAITFLAG and INT_SETFLAG functions. System flags are allocated by GENCPM, and are located in the SYSDAT segment. Only one process at a time may wait on a particular system flag, and only one interrupt service routine may set a particular flag. If a process is waiting on a flag, a second ?WAITFLAG operation by another process specifying the same flag returns an error. Similarly, if a flag is already set by an interrupt service routine, another INT_SETFLAG operation to the same flag returns an error. Table 4-1 shows the register conventions for ?INT_SETFLAG; Appendix B shows the BIOS Kernel conventions for ?WAITFLAG.

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_CHARSCAN function with each character received from the physical device. This physical device is usually the system or a remote console, and the INT_CHARSCAN function 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 service routine in progress. Exiting with an IRET prevents 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, an interrupt cannot preempt a running interrupt service routine. In this case, the interrupt service routine can always exit by executing 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. Listings 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:

```
CSEG
extrn   @SYSDAT:word
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. You can force a code segment override by declaring @SYSDAT an external within the code segment, as shown above.

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 CALLF to INT_SETFLAG 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. Disable interrupts and switch back to the original stack.

9. Ensure interrupts are disabled 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, execute a 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 interrupts 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

Format: Function
Explanation
Entry Registers
Exit Registers

INT_SETFLAG

Call Far to this routine to signal the occurrence of a hardware event.

Entry Registers: DL = flag number to set
DS = SYSDAT (BIOS data segment)

Exit Registers: AX = 0 successful operation
AX = 0FFFFh then error and
CX = 4 flag number out of range
CX = 5 flag already set (flag overrun)
(In either case: 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.

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 software 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 function 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 software CPU loops for these time-outs, then replaced with calls to ?DELAY as one of the last steps in the BIOS implementation.

EOF

CPMPIG5.WS4 (= "CP/M-86 Plus Installation Guide", section 5)

(Edited by Emmanuel ROCHE.)

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 CPMP.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 Kernel Code Header, a JMP BIOSINIT instruction starts the BIOSINIT routine. Section 11 discusses 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; otherwise, it is equal to 0. 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 current CONIN: device is interrupt-driven.

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 disk boot 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 already described, 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 unknown locations. 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 set the specific interrupt vectors they need. All interrupt vectors should be initialized when BIOSINIT returns to the BDOS. However, during debugging you usually leave several interrupt vectors unchanged to allow CP/M-86 1.X and DDT-86 to monitor your CP/M-86 Plus BIOS. Section 10 examines debugging.

The ?INIT routine 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 routines associated with 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. Only one of the DPH_INIT routines (see Section 7) should initialize the disk controller in this case.

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 ?WAITFLAG and INT_SETFLAG 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 the required number of system flags are indeed available, and halt initialization if they are not. GENCPM sets @BH_NEXTFLAG and @BH_LASTFLAG at system generation time.

EOF

CPMPIG6.WS4 (= "CP/M-86 Plus Installation Guide", section 6)

(Edited by Emmanuel ROCHE.)

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. A third 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 character I/O devices in the module CHARIO.A86. The CHARIO.A86 file on the distribution disks is a useful reference while reading this section.

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 disks.

Listing 6-1. Character Device Block Format

```
;*******************************************************
;
;     Console Device Block Equates
;
;*******************************************************
;
;     +------+------+------+------+------+------+------+------+
; 00h: |            NAME             |  SUPCHAR  |
;     +------+------+------+------+------+------+------+------+
; 08h: |  CURCHAR  |SUPOEM|CUROEM| TXB  | RXB  | TYPE |IINPUT|
;     +------+------+------+------+------+------+------+------+
; 10h: |NFLAGS|RESVD |   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_NFLAGS      equ     byte ptr 16
CDB_RESVD       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, and CRT0_CT values are equates defined in CHARIO.A86. The symbols CRT0_INIT, CRT0_INPUT, CRT0_INSTAT, CRT0_OUTPUT, and 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 describes each CDB field:

Table 6-1. Character Device Block Data Fields

Format: Data Field
Explanation

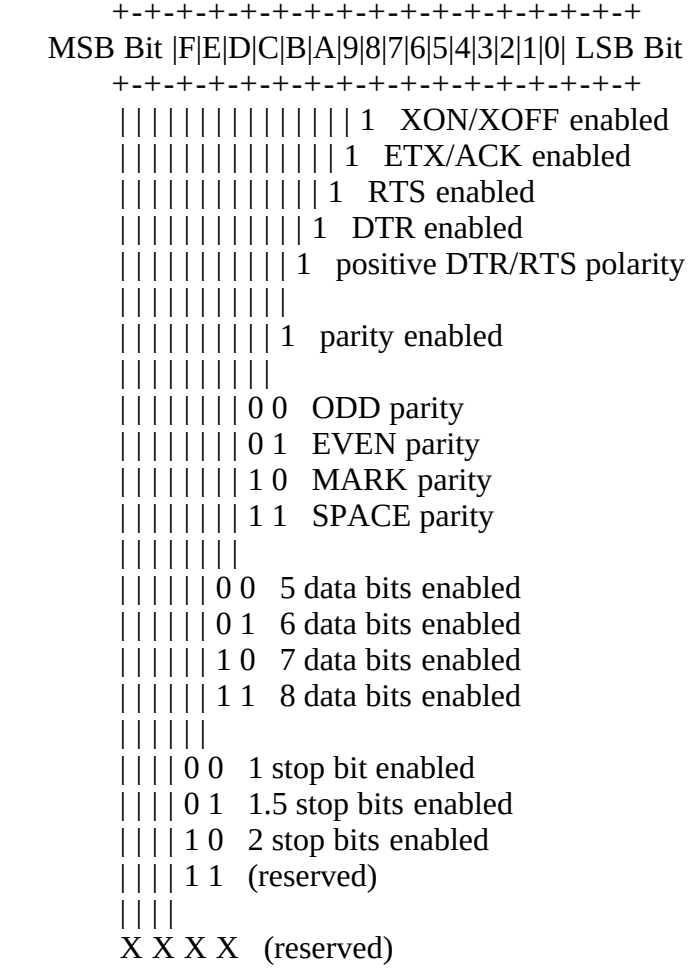
CDB_NAME [Six-character name of this physical device]
The device name must be in capital alphanumeric ASCII characters, left-justified in the field, and padded on the right with ASCII spaces.

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 stop 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:

```
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 MSB Bit |F|E|D|C|B|A|9|8|7|6|5|4|3|2|1|0| LSB Bit
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       | | | | | | | | | | | | | | | | 1   XON/XOFF supported
       | | | | | | | | | | | | | | | | 1   ETX/ACK supported
       | | | | | | | | | | | | | | | | 1   RTS supported
       | | | | | | | | | | | | | | | | 1   DTR supported
       | | | | | | | | | | | | | | | | 1   DTR/RTS polarity supported
       | | | | | | | | | | | |
       | | | | | | | | | | | | 1   ODD parity supported
       | | | | | | | | | | | | 1    EVEN parity supported
       | | | | | | | | | | | | 1    MARK parity supported
       | | | | | | | | | | | | 1    SPACE parity supported
       | | | | | | | |
       | | | | | | | | | | | | 1   5 data bits supported
       | | | | | | | | | | | | 1   6 data bits supported
       | | | | | | | | | | | | 1   7 data bits supported
       | | | | | | | | | | | | 1   8 data bits supported
       | | | |
       | | | | | | | | | | | | 1   1 stop bit supported
       | | | | | | | | | | | | 1   1.5 stop bits supported
       | | | | | | | | | | | | 1   2 stop bits supported
```

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.



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 if bit 4 of the CDB_CURCHAR field is 0, and positive if bit 4 of the CDB_CURCHAR field is 1.

The DEVICE utility alters this field to change the current device characteristics, and 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.

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 (described later in this table) has the CI_SOFTBAUD bit set, the DEVICE utility can change the baud by first setting the CDB_TXB and CDB_RXB fields, then calling the CDB_INIT routine. See the discussion on the CDB_INIT routine in "Character Device Block (CDB) Routines" later in this section, 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

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_INTCONIN in the BIOS Kernel Data Header. The @BH_INTCONIN 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_NFLAGS
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_RESVD
This field is unused by the BDOS or the BIOS Kernel.

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 set 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 in setting the device to these specifications, or it returns AX = 0FFFFh if there is an error. On entry, DS:BX specifies the address of the CDB for this device. "Character Device Block (CDB) 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.

CDB_OUTSTAT
This is the offset address of the character output status routine for this device. 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 of 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 (Near Return instruction).

The CDB routines 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 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).

Table 6-2. CDB_ Character I/O Routines

Format: Routine
    Explanation

CDB_INIT

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 makes a IO_DEVINIT call after changing the protocol or baud of the device; in other words, DEVICE makes the call when it changes 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 0 before calling CDB_INIT, and the Kernel IO_DEVINIT function sets DL to 1 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, except 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
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

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_OUTPUT

The CDB_OUTPUT 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

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 the 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 character input or character output can be interrupt-driven, or both can be. 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 to 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 at the end of this subsection 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 disks.

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_INTCONIN field in the BIOS Kernel Data Header. The value in @BH_INTCONIN informs the BDOS that console input is interrupt-driven. When @BH_INTCONIN 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. 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. Section 4 shows the register conventions for the INT_CHARSCAN function.

In Listing 6-3, the input interrupt handler 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. In such instances, it is usually desirable to discard 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. (CRT0_INPUT 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 the 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. (Section 4 shows the register conventions for the BDOS INT_SETFLAG function.)

In Listing 6-3, this "flag waiting" variable is 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 of this 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. The SYSDAT.LIB file contains equates for the INT_SETFLAG and INT_CHARSCAN functions. The hardware equates for the CompuPro port addresses and programmable interrupt controller (PIC) can be found in the PIC.LIB and the CHARIO.A86 files.

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 instruction.

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

        extrn   ?waitflag:near          ;routine in Kernel
        extrn   @sysdat:word            ;in Kernel code segment

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 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 device
          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 char
        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 out
        dec BUF_COUNT[bx]           ;one less char in buffer
        sti                         ;enable CPU interrupts
        ret

crt0_input_int:     ;Input interrupt service routine for the 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               ;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 altered
        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 char
            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_done1
          mov BUF_FWAIT[bx],0       ;yes - set the flag
          push dx                   ;?SETFLAG alters AX,BX,CX,DX
          mov dl,BUF_FLAGN[bx]      ;DL=flag for this input device
```

```
        callf int_setflag
        pop dx
cii_done1:
      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 routine

      DSEG

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

;     console input interrupt 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
```

Character Output Interrupt

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 at the end of this subsection shows an example implementation of buffered interrupt-driven character output. It is also found in the CHARIO.A86 file on the distribution disks.

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 CDB_OUTPUT routine must call ?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 that 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 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. The SYSDAT.LIB file contains equates for the INT_SETFLAG and INT_CHARSCAN functions. The hardware equates for the CompuPro port addresses and programmable interrupt controller (PIC) can be found in the PIC.LIB and the CHARIO.A86 files.

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 instruction.

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

        extrn   ?waitflag:near          ;routine in Kernel
        extrn   @sysdat:word            ;in Kernel code segment

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 buffer
        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 interrupts
```

```
        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 0
            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
                push cx ! push dx       ;?SETFLAG alters AX,BX,CX,DX
                mov dl,BUF_FLAGN[bx]    ;then set the flag
                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 interrupt
        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 interrupt 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 buffer
                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 what action should be taken. Usually, the choices given to 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 upon 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 that the application does not want the screen altered, and you should display messages only for catastrophic errors. The "Programmer's Guide" describes the File System Error Mode, which 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 Appendix C.) If the process's File System Error Mode byte is equal to 0FFh, the process is in Return Error Mode, and most error messages should not be displayed.

EOF

CPMPIG7.WS4 (= "CP/M-86 Plus Installation Guide", section 7)

(Edited by Emmanuel ROCHE.)

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 is a short discussion of BIOS disk I/O error messages.

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 performance 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 disk boot loader and CPMLDR, which read the CPMP.SYS file into memory. These tracks are called the boot tracks. This area is optional, and is needed only if the disk boots the system. For example, a hard disk not used for boot operations has no boot tracks.

The second area is the directory, and starts immediately after the boot 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. You define 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 files.

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

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

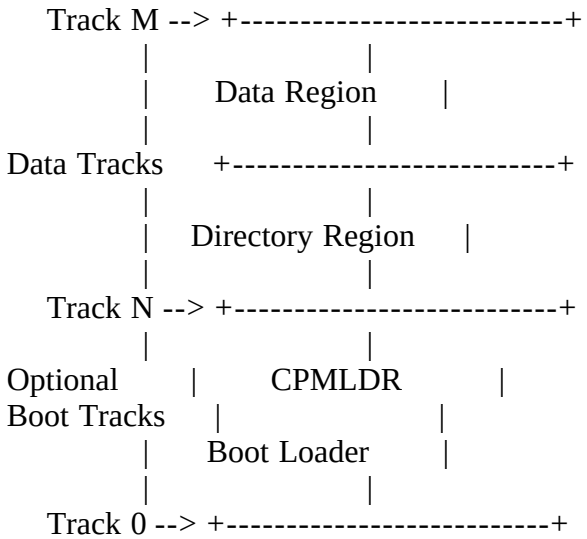


Figure 7-1. CP/M-86 Plus Disk Organization

In Figure 7-1, the first N tracks are the boot 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 disks should have two boot 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, systems 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 an information structure.

Listing 7-1, contained in the file DISK.LIB on the distribution disks, 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, eight-inch disk. (The S1DSM symbol in the listing is defined in the DISKIO.A86 file as the number of allocation blocks on a single-sided, single-density disk.)

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      1100$0000b  ;alloc0 - 2 directory blocks
            db      0000$0000b  ;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

Format: 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, which are described later in this table.

DBP_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. (Logical records are 128 bytes long as defined by the file-related system calls.) 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.

The allocation block size is the minimum allocation unit for file I/O under CP/M-86 Plus. Larger block sizes waste more space at the end of a sequential file and throughout a random file. But larger block sizes require less directory space to represent large files, and allow for quicker access to the file's data records. The available allocation block sizes are shown below, along with the corresponding DPB_BSH and DPB_BLM values

Block Size	DPB_BSH	DPB_BLM
1,024	3	7
2,048	4	15
4,096	5	31
8,192	6	63
16,384	7	127

DPB_EXM [Extent mask]
The extent mask determines the maximum number of 16 Kbyte logical extents that is contained in a single directory entry. It is determined by the allocation block size and the number of allocation blocks the drive contains, as shown in the following information. Note that you cannot have a block size of 1 Kbyte on a disk containing 256 or more blocks, since one directory entry would only represent 8 Kbytes, and not an entire extent. In this latter case, you must

use a larger block size.

Block Size	If Number of Blocks < 256, then DPB_EXM =	If Number of Blocks >= 256, then DPB_EXM =
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.

DPB_DRM [Directory maximum]
The directory maximum defines the total number of directory entries on this drive. Allocation blocks are reserved for the directory by the DPB_AL0 and DPB_AL1 fields discussed below. DPB_DRM is 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 following table shows the number of directory entries for one allocation block and for 16 blocks using the available block sizes.

Block Size	Directory Entries Per Block	Directory Entries Per 16 Blocks
1,024	32	512
2,048	64	1024
4,096	128	2048
8,192	256	4096
16,384	512	8192

Choose directory size carefully. Once CP/M-86 Plus writes on disks, the directory size cannot be changed, though the disk can be copied to another disk with a larger directory.

DPB_AL0, DPB_AL1 [Directory allocation vector]
DPB_AL0 and DPB_AL1 reserve from 1 to 16 allocation blocks for the directory. The directory is contiguous starting with block 0. The directory allocation vector is a bit map with each bit representing an allocation block being used for the directory. Table D-4 in Appendix D shows the legal values of DPB_AL0 and DPB_AL1, based on the number of allocation blocks desired for the directory.

DPB_CKS [Checksum vector size]
A checksum vector is required for removable media, in order to ensure the data integrity of the disk system. The lower order 15 bits of DPB_CKS determine the length of the directory checksum vector addressed by the Disk Parameter Header (see the next subsection). These 15 bits also determine the number of directory entries the BDOS will checksum when a drive is "logged in". (The process of logging in a drive is discussed in "Detecting Media Changes" latter in this section, and under "Drive Status" in Section 3 of the "Programmer's Guide"). Each byte of the checksum vector is the checksum of 4 directory entries.

The high-order bit, when set in the DPB_CKS field, indicates a drive containing permanent or nonremovable media. Ensuring data integrity on permanent media drives requires fewer directory accesses, and allows the buffering of more data in memory, requiring fewer disk writes. The result is that permanent media drives can get up to 30% better performance than removable media drives. Typically, hard disk systems have the DPB_CKS value of 8000h, indicating no checksumming and permanent media.

In systems that can detect the door open for removable media drives, checksumming is only done when the DPH_DOPEN byte in the DPH is set to 0FFh, indicating that the drive door has been opened. The drive is thus treated as a permanent media drive until the drive door is opened. The high-order bit in DPB_CKS is set in this case, and the low-order 15 bits reflect the required checksum vector size. "Detecting Media Changes" later in this section covers this topic in more detail.

DPB_OFF [Track offset]
The track offset is the number of boot tracks at the beginning of the disk. DPB_OFF is equal to the track number on which the directory starts. Using this field, more than one logical disk drive can be mapped onto a single physical drive. See "Multiple Logical Drives" later in this section for more information.

DPB_PSH [Physical record shift factor]
The physical record shift factor is used by the BDOS to calculate the physical sector from the logical record number. The logical record number is shifted the number of DPB_PSH bits to the right to calculate the physical record. (Logical records are 128 bytes long as defined by the file-related system calls.)

DPB_PHM [Physical record mask]
The physical record mask is used by the BDOS to calculate the logical record offset within a physical sector by masking the logical record number with the DPB_PHM value. The following table shows the DPB.

Physical Sector Size	DPB_PSH	DPB_PHM
128	0	0
256	1	1

512	2	3
1024	3	7
2048	4	15
4096	5	31

Disk Parameter Header (DPH)

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 disks.

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 |NFLAGS|
;     +-----+-----+-----+-----+-----+------+------+

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_NFLAGS      equ     byte ptr 30
```

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

distribution disks. The symbols XLTD3 and DPBD6 define the offsets of the translation table and the DPB. The symbols beginning with FD_ are the offsets of the disk I/O routines for the logical drive represented by this DPH.

Listing 7-4. Disk Parameter Header 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 parameter table
            dw      0FFFFh      ;checksum
            dw      0FFFFh      ;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

Format: 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 0 by the INIT routine or the load image.

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 that 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 disk, 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 similar to 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 that 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 DPB_CKS in the DPB is 0 or 8000h, no checksum area is used, and DPH_CSV can be 0. Values for DPB_DRM and DPB_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.

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 of 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.

DPH_DIRBCB
This field contains the offset of the Directory Buffer Control Block (DIRBCB) Header. The DIRBCB Header contains 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 a different 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 (DATBCB) 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 sector 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.

DPH_HSHTBL

This field contains the paragraph address of the optional directory hash table (HSHTBL) associated with a logical drive. The BDOS assumes that the hash table offset address to be 0. If you decide not to use directory hashing to save memory space, set DPH_HSHTBL to 0. 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. Each entry in the hash table must contain four bytes for each directory entry of the disk. If this field is 0FFFFh, GENCPM initializes DPH_HSHTBL, 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 for 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.

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 2 and 4 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 0 in DPHA and DPHB. Since drives C: and D: use the second controller, the DPH_CHNNL fields in DPHC and DPHD are set to 1. Only the BIOS uses this field.

DPH_NFLAGS
The DPH_NFLAGS byte contains the number of system flags used by this drive. If more than one drive shares 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 0 in their DPH_NFLAGS fields. The first DPH of several that share a controller can be identified by a DPH_CHNNL value of 0. GENCPM uses DPH_NFLAGS 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 disks. 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 | DRIVE | MCNT  |   Drive and Multisector Count
;         +-------+-------+
;      +12 |     TRACK     |   Track number
;         +-------+-------+
;      +10 |    SECTOR     |   Physical sector number
;         +-------+-------+
;       +8 |    DMA_SEG    |   DMA segment
;         +-------+-------+
;       +6 |    DMA_OFF    |   DMA offset
;         +-------+-------+
;       +4 |    RET_SEG    |   BDOS return segment
;         +-------+-------+
;       +2 |    RET_OFF    |   BDOS return offset
;         +-------+-------+
;     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

Format: Data Field
Explanation

IOPB_DRIVE [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 16 Kbytes directly to or from the DMA address in a single operation. Note that 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.

IOPB_TRACK

The IOPB_TRACK defines the track for the specified drive to seek. The BDOS defines IOPB_TRACK relative to 0. For disk hardware which defines track numbers beginning with a physical track of 1, 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 0, so for disk hardware which defines sector numbers beginning with a physical sector of 1, 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.

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 copy the data between the DMA address and a local buffer accessible to the controller. (DMA is an acronym for Direct Memory Address, a term used in the context of the BIOS for disk I/O operations which transfer physical sectors directly to memory, and vice versa.)

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 that 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 0.

Table 7-4. DPH_Disk I/O Routines

Format: Routine
Explanation

DPH_INIT
The DPH_INIT routine initializes the hardware associated with a particular drive. BIOSINIT calls the DPH_INIT routine for each DPH defined in the BIOS. 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: 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 = offset of DPH if no error
BX = 0 if error
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 be reinitialized. If the type of drive changes, the DPH_DPB field is changed to point the DPB defining the new drive. "Automatic Density and Side Selection," which appears later in this section, discusses the DPH_LOGIN routine in more detail.

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 1 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.

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 1 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 multiple 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 0.

Listing 7-6 after this table shows a simple implementation of a multisector read/write routine that performs single sector operations until all the sectors are transferred. The DISKIO.A86 module on the distribution disks contains a read/write routine that performs multisector transfers at the controller level. The RW64.A86 file provides another example showing a multisector read/write routine that cannot transfer across a 64-Kbyte page boundary, because of hardware restrictions.

In Listing 7-6, if IOPB_MSCNT is 0, 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. If there is no error, the IOPB_MSCNT value is decremented. When IOPB_MSCNT equals 0, 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 0. 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

```
        include disk.lib

maxsec  equ     8               ;sectors per track (example)
secsiz  equ     512             ;sector size (example)

hd_io:          ;common code for disk read and writes
```

```
;-----
;       Entry:  BP = IOPB on stack
;               RFLAG = true if reading, else writing
;       Exit:   AL = 0 if success
;               AL = 1 if error
;
; Use IOPB to form a series of single sector read or write
; operations.

        push es                 ;save process environment
        mov al,1                ;error return
        cmp iopb_mcnt,0         ;if Multisector Count = 0
        je return_rw            ;return error

hdio1:
         call iohost            ;read/write physical sector
         or al,al               ;test for error
         jnz return_rw          ;return error
         dec iopb_mcnt          ;decrement Multisector Count
         jz return_rw           ;if multisector = 0 return
           mov ax,iopb_sector
           inc ax               ;next sector
           cmp ax,maxsec
           jb same_track        ;is sector < sectors per track
             inc iopb_track     ;no - next track
             xor ax,ax          ;initialize sector to 0
same_track:
           mov iopb_sector,ax   ;save sector #
           add iopb_dmaoff,secsiz ;increment DMA offset by
                                  ;sector size
           jmps hdio1           ;read/write next sector
return_rw:
        pop es                  ;restore process environment
        ret                     ;return with error code in AL

iohost:         ;single physical sector read/write
;------
;       entry:  BP = IOPB on stack
;               RFLAG = true if reading, else writing
;       exit:   AL = 0 if success
;               AL = 1 if error
;               DS preserved
;
; Transfer one physical sector as indicated by the IOPB
; parameters IOPB_SECTOR, IOPB_TRACK, IOPB_DRIVE to or from
; IOPB_DMASEG:IOPB:DMAOFF.
;
;       Your hardware-dependent single sector transfer
;       routine goes here.

        ret

DSEG
        rflag   rb      1
```

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 interrupts on removable media drives can be detected to improve disk I/O performance and improve data integrity. 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. Other modifications include the support of skewed disk formats for compatibility with media written from other machines or operating systems, and the implementation of a "memory disk".

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:

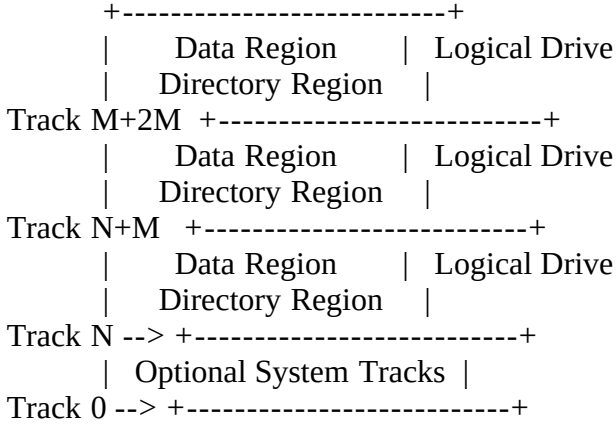


Figure 7-2. Multiple Logical Drives

Figure 7-2 shows three logical drives mapped onto one physical drive. Three separate DPBs 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

Disk drives under CP/M-86 Plus are classified whether the media they contain is permanent or removable. "Removable media drives" support media changes; "permanent media 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 disks.

If a drive's media is permanent, the BDOS always accepts the contents of the physical sector buffers as valid. In addition, it also accepts the results of hash table searches on the drive.

On removable media 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_HSHTBL field. (These fields and data areas are usually set and allocated automatically by GENCPM.)

Before the BDOS performs certain directory-related functions, it verifies that 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 media 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 media drives, as compared to permanent media drives. In addition, the protection provided by classifying a drive's media 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 data on the new disk will be overwritten.

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 BDOS to treat the drive as if it contained permanent media until the occurrence 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 checksum 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, then 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 that 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 disk. If the BIOS can detect the media type, automatic (auto) 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 you. 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. Additionally, 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 which you supply 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_SELDSK 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 that 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-sector transfers, it is more difficult to optimize transfer time for multisector operations. One method of doing this is to have the BIOS read/write 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.

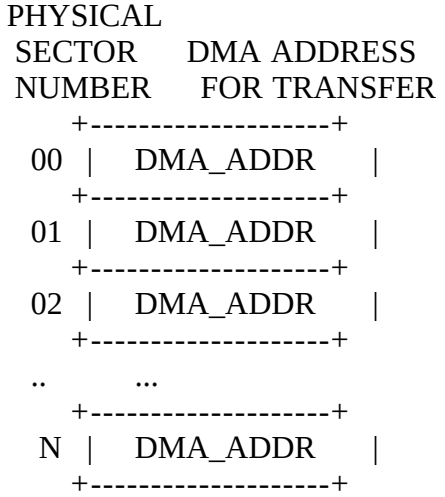


Figure 7-3. DMA Address Table for Skewed Multisector I/O

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 can 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 disk parameters are taken from the DPH and DPB to be stored in local variables. Once the physical sector 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

```
        include disk.lib

        CSEG

rw_skew:        ;unskews for reads/writes of multiple sectors
;-------
;       entry:  SI = offset of read or write routine
;               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 sector size
        mov secsiz,ax           ;and save it
        call initdmatbl         ;initialize DMA offset table
        cmp iopb_mcnt,0
        je ret_error
rw_1:
         mov ax,iopb_sector     ;is sector < sectors per 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
```

```
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 Multisector Count
        jnz rw_1            ;if IOPB_MCNT<>0 store next
                          ; sector DMA
rw_sects:                   ;read/write sectors in DMA table
      mov al,1              ;preset error code
      xor bx,bx             ;initialize sector index
rw_s1:
      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,iopb_track   ;get track # from IOPB
        mov itrack,ax
        mov isector,bx      ;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 & index
        or al,al! jnz err_ret ;if error occurred return
no_rw:
      inc bx                ;next sector index
      cmp bx,maxsec         ;if not end of table
      jbe rw_s1             ;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
      inc cx                ;may 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

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

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

        DSEG

isector rw      1       ;parameters for single sector
itrack  rw      1       ;read/write operation
idmaoff rw      1
idmaseg rw      1

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 Kbytes 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 (RAMdisk) 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 Kbytes 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 Kbytes 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

```
        include disk.lib

mdiskbase       equ     0C000h  ;base paragraph
                        ;address of M: disk
        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 16 Kbytes
          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:   AL = 0  (always successful)

; 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:   AL = 0  (always successful)
; 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 number
        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,iopb_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 the system. (A physical sector is the sector size as defined by the DPB for the drive.) Since GENCPM generates the data structures, and initializes the fields in the DPH for disk buffering, this material is optional.

The BDOS uses Buffer Control Blocks (BCBs) to locate and manage physical sector buffers. A 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     ;offset of first DIRBCB
            db      0ffh        ;used by BDOS
```

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 (DIRBCB)

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 accommodate 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_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               ;Write Pending
        rw      2               ;Track, Sector
        dw      dirbuf0         ;Buffer Offset
        dw      dirbcb1         ;BCB Link
        dw      0               ;Reserved
```

Table 7-5 defines the DIRBCB fields:

Table 7-5. DIRBCB Data Fields

Format: 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 all other BCB fields when a BCB and its buffer are used.

BCB_RECORD
The BCB_RECORD number identifies the first logical record contained in the BCB buffer. Since the size of the BCB buffer is a physical sector, it can contain several 128-byte logical records. The logical record number is relative to the beginning of the logical drives, where the first record of the directory is logical record number 0.

BCB_WFLG [Write pending flag]
The BDOS sets the BCB_WFLG to 0FFh to indicate that the buffer contains unwritten data. When the data is written to disk, the BDOS sets the BCB_WFLG to 0.

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 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. Thus, BCB_SECTOR is usually defined to be the same as the physical sector number.

BCB_BUFOFF
For DIRBCBs, this field equals the offset address of the buffer within SYSDAT.

BCB_LINK
The BCB_LINK field contains the offset address of the next BCB in the linked list, or 0 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 that 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 (DATBCB)

Listing 7-12 shows the format of the Data Buffer Control Block (DATBCB):

Listing 7-12. Data Buffer Control Block (DATBCB)

```
;*******************************
;*
;*   DATBCB Format
;*
;*******************************
;
;     +------+------+------+------+------+------+------+------+
; 00h: | DRV |      RECORD      | WFLG | 00h |    TRACK    |
;     +------+------+------+------+------+------+------+------+
; 08h: |   SECTOR    |   BUFSEG   |   LINK    |   RESERVED  |
;     +------+------+------+------+------+------+------+------+

BCB_DRV     equ     byte ptr 0
BCB_RECORD  equ     byte ptr 1
BCB_WFLG    equ     byte ptr 4
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_BUFSEG 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 the Transient Program Area (TPA), and must be on a paragraph boundary.

DPH_HSHTBL and BCB_BUFSEG Initialization

The hash table address for a particular logical drive is a paragraph address kept in the DPH_HSHTBL field. The address of the data buffer associated with a DATBCB is also a paragraph address. If you define the hash tables or the data buffers in the BIOS, you must "fix-up" these addresses to be paragraph address values at BIOS initialization time. The following code fragment accomplishes this, and can be made part of your INIT module. Note that GENCPM automatically sets the paragraph address of the hash tables and data buffers it creates in the appropriate DPH_HSHTBL and BCB_BUFSEG fields.

Listing 7-13. DPH_HSHTBL and BCB_BUFSEG Initialization

```
; Initialize DPH_HSHTBL and BCB_BUFSEG fields. The hash
; table and data buffers must be paragraph aligned. This
```

```
; code fragment fixes up the offset addresses in DPH_HSHTBL and
; BCB_BUFSEG to be paragraph addresses. 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 paragraph 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.

        include disk.lib

        CSEG
        extrn   @bh_dphtable:word       ;in BIOS Kernel
        extrn   @dpha:word              ;in DISKIO module

BCB_BUFSEG      equ     word ptr 10     ;DATBCB fields
BCB_LINK        equ     word ptr 12

        mov cx,16                       ;16 maximum drives
        xor si,si                       ;SI = 0
hash_init:
        push cx                         ;save drive count
        mov bx,@bh_dphtable[si]         ;BX = next DPH address
        test bx,bx
        jz next_dph                     ;if not 0, BX = DPH
          mov ax,DPH_HSHTBL[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_HSHTBL[bx],ax       ;make the fixup

next_dph:
        pop cx                          ;restore the drive count
        add si,2                        ;index for next DPH offset
        loop hash_init

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

        mov bx,offset @dpha             ;DPHA from DISKIO module
        mov bx,DPH_DATBCB[bx]           ;BX=DATBCB header
        mov bx,[bx]                     ;BX=DATBCB
next_datbcb:
        mov ax,BCB_BUFSEG[bx]           ;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                       ;paragraph address
        mov BCB_BUFSEG[bx],ax           ;make fixup
```

```
    mov bx,BCB_LINK[bx]          ;BX=next BCB
    or bx,bx                     ;0 if end of linked list
    jnz next_datbcb

; ... the rest of your initialization code ...
```

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
```

Note that d: is 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 filename and filetype.

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 is made to write to a drive set to Read-Only through the DRV_SETRO system call. If the 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 changing or translating the BDOS messages.

If you plan to display more information about a specific hardware error, the discussion in "Character I/O Error Messages" in Section 6 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 that 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 Appendix C.) If the process's File System Error Mode byte is equal to 0FFh, the process is in Return Error Mode, and most error messages should not

be displayed.

EOF

CPMPIG8.WS4 (= "CP/M-86 Plus Installation Guide", section 8)

(Edited by Emmanuel ROCHE.)

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 the 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. The DS register on entry to the interrupt service routine must be on the stack when a JMPF to INT_DISPATCH is made. INT_DISPATCH is the double word address of the process 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 update the system time of day structure once per second. 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, as well as 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 that 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, you may need to supply a utility that would replace the DATE utility for setting 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 CLOCK.A86 file on the distribution disks. The equates for the Programmable Interrupt Controller and the SYSDAT variables are from 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 more accuracy over long periods of time.

Section 4 discusses the general structure of an interrupt service routine under CP/M-86 Plus. Note that the TICK_INT routine in Listing 8-1 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             ;nonspecific end of interrupt
MASTER_PIC_PORT equ     50h
SLAVE_PIC_PORT  equ     52h

;include sysdat.lib                    ;contains the following equates

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     ;get BIOS data segment
        inc @bh_inint                   ;signal executing interrupt handler
```

```
        mov saveax,ax
        dec tick_cnt                ;tick count
        jnz cont_tick               ;if second not yet up, branch to exit
          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
          pop ds                    ;go back to incompleted
          iret                      ;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


EOF
```

CPMPIG9.WS4 (= "CP/M-86 Plus Installation Guide", section 9)

(Edited by Emmanuel ROCHE.)

Section 9: System Generation

This section describes the procedures necessary to generate the CP/M-86 Plus system contained in the CPMP.SYS file. The CompuPro BIOS modules on the distribution disks are used for specific examples. Note that the number of, and names for, the BIOS modules for your machine are likely to differ from the examples.

Generation of the CPMP.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 all 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 CPMP.SYS from the BIOS3.SYS, BDOS3.SYS, and optionally, the CCP.CMD files.

ASSEMBLING THE BIOS MODULES

The following RASM-86 commands assemble the example BIOS modules on the distribution disks. The "Programmer's Utilities Guide" documents RASM-86.

```
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, DISKIO.OBJ, and CLOCK.OBJ. Similarly, you must also assemble each of your BIOS modules. If you are debugging a particular module, use the RASM-86 $LO option 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 public and external declarations. It takes the following command line form. The arguments in brackets are optional.

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

MODEDIT allows new device drivers to be added to an existing 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 in the 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, @CDBD, and @CDBA appear in the file CHAR1.OBJ in the same order as they appear in this sentence. MODEDIT changes these symbol names to @CDBA, @CDBB, and @CDBC respectively. If CHAR2.OBJ contains the public symbols in the order @CDBA then @CDBB, MODEDIT renames the symbols to @CDBD and @CDBE respectively. This occurs because @CDBA, @CDBB, and @CDBC were used for the first three Character Device Blocks in the CHAR1.OBJ file. Also, notice that the symbol names in the different modules, or within the same module, do not have to be unique. In this example, the symbol name @CDBA occurs in both files, MODEDIT changes the two @CDBA symbols to the unique names @CDBC and @CDBD respectively. 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. The following command links the example BIOS modules. The "Programmer's Utilities Guide" describes LINK-86 in greater detail.

```
A>LINK86 BIOS3.SYS = BIOSKRNL,INIT,CHARIO,DISKIO,CLOCK,ZERO.L86
  [DATA[ORIGIN[0F00]], SEARCH]
```

All of your BIOS modules OBJ files must be present in similar commands that create the BIOS3.SYS file. Since the BIOS data starts at 0F00h, the ORIGIN

option must be present in the LINK-86 command when creating the BIOS3.SYS file. This usage of the ORIGIN option assumes there are no ORG statements in your BIOS modules. (The BIOS Kernel starts the code and data segments with an ORG 0000h statement.) 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 that 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.

Generally the LINK-86 command takes the following form:

```
A>LINK86 BIOS3.SYS = BIOSKRNL,MOD1,MOD2,...MODN,ZERO.L86
  [DATA[ORIGIN[0F00]], 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, the LINK-86 command would take the following form:

```
A>LINK86 BIOS3.SYS = BIOSKRNL,INIT,CHARIO,DISKIO,HDISKIO,
  CLOCK,ZERO.L86 [DATA[ORIGIN[0F00]], SEARCH]
```

The LINK-86 INPUT option is helpful when you are repeatedly generating a BIOS3.SYS file during development. The INPUT option allows the command tail to be read from a file. For instance, if you place the command tail in 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 CPMP.SYS. This file becomes the memory resident part of the CP/M-86 Plus operating system. You must read CPMP.SYS into memory at a specific location, then transfer control to it. GENCPM runs under either CP/M-86 1.X, CP/M-86 Plus, Concurrent CP/M, or MP/M-86.

GENCPM builds the CPMP.SYS file from the files BDOS3.SYS, BIOS3.SYS, and optionally, CCP.CMD. You can 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 use by 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 hexadecimal unless it is preceded by a pound sign (#), which indicates that the numeric is decimal. You can answer any question either in hexadecimal 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 CPMP.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 these screens relates to a single topic such as disk buffer allocations. In this section, each screen appears as a figure, followed by an explanation for each question displayed in the screen. Note that the default values shown in the GENCPM screens used in this section were chosen for tutorial purposes and are not meant to be used to generate a working CPMP.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 "Use GENCPM.DAT file for defaults" 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 do not directly alter the CPMP.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.

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 CPMP.SYS file. These initial questions cannot be repeated after you enter a Y.

Enter N - GENCPM repeats the previous questions, and displays your previous

input in the parentheses, so you can correct 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 categories, 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

The following explains the 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 return to the Main Menu. When you change a value in the sub-menus, the new value appears in parentheses, and becomes the default value 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. The answers you supply in response to this screen inform GENCPM about files it needs to find, create, or delete.

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

Create a new GENCPM.DAT file                    (N)?
Destination drive                               (A:)?
Delete (instead of rename) old CPMP.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 CPMP.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 CPMP.SYS file is placed in the current default user area of the destination drive.

Question Variable: DESTDRV

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

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

Enter Y - GENCPM deletes the existing CPMP.SYS file, and creates a new CPMP.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 require a CCP.CMD file on disk. However, the memory area occupied by the operating system is larger, since it includes the CCP.

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 (N)?

This question only affects the behavior of the C_READBUF system call. The rubout character (DEL, 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 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 placed in ROM. See Appendix G for more information on placing CP/M-86 Plus in ROM.

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 1 Kbyte 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 reduces the memory specified by this menu, to define the memory available for loading transient programs. This remaining memory is called the Transient Program Area (TPA). GENCPM initializes the Memory Descriptor 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 thru 1FFF:0, the second partition skips one paragraph and specifies memory from 2001:0 thru 2FFF:0, inclusive. Because these two memory areas are noncontiguous, the BDOS cannot coalesce 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 that the total number of Memory Descriptors available to describe memory fragmentation during system operation is limited to 32. See "Memory Management" in Section 5 of the "Programmer's Guide".

Question Variable: MEMPART# (where "#" is in the range 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       030CH       ( 08 ,   04 ,    Yes )?
 B:    0040H       0000H       ( A: ,   A: ,    Yes )?
 C:    0040H       0000H       ( A: ,   A: ,    Yes )?
 D:    0040H       0145H       ( 03 ,   02 ,    Yes )?
 E:    0040H       0104H       ( 04 ,   D: ,    Yes )?

   Accept new buffer definitions (Y)?
```

Figure 9-7. GENCPM 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_HSHTBL, DPH_ALV, DPH_CSV, 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 or DPH_CSV, or both, are set to 0FFFFh, GENCPM automatically calculates and reserves the amount of memory the drive requires for allocation and checksum vectors. DPH_ALV and DPH_CSV are set to the offset of the appropriate vector. GENCPM displays no questions or messages when it automatically creates allocation and checksum vectors.

DPH_HSHTBL 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_HSHTBL 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 BCB Header 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 CPMP.SYS file. Uninitialized areas are reserved for the operating system, but are not made part of the CPMP.SYS file. This keeps the CPMP.SYS file size to a minimum, and improves system boot time. Appendix E contrasts the CPMP.SYS file and the memory image of CP/M-86 Plus.

GENCPM assumes that 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. The "Memory Allocated" column in this screen reflects only the sector buffers and the BCBs allocated, not any other space allocated for the drive's hash table, checksum vector, or allocation vector. (A 16 byte BCB is required for each sector buffer.)

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 CPMP.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  121BH
   1     2001H  0FFFH

CPMP.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 important if you are placing CP/M-86 Plus in ROM (see Appendix G). The System Code is the segment address and length 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 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 to eliminate 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 Plus prompt. GENCPM does not modify any existing CPMP.SYS or GENCPM.DAT files.

Example GENCPM.DAT File

The following shows the contents of a GENCPM.DAT file that can be used to generate a CPMP.SYS file from the example BIOS on the distribution disks. A GENCPM.DAT file is also included on the distribution disks.

Listing 9-1. Example GENCPM.DAT File

```
CRTDATF  = N
```

```
CLRSCR   = 1B, 45
HOMCSR   = 1B, 48
DESTDRV  = B:
DELSYS   = N
CCPYES   = Y
BACKSPC  = N
RUBOUT   = Y
PAGWID   = 50
PAGELEN  = 18
BOOTDRV  = A:
TICKS    = 3C
ADDFLGS  = 00
OSBASE   = 0040
OSDBASE  = 0000
ADDMEM   = 0000
MEMPART0 = 0040, 1FC0
MEMPART1 = 0000, 0000
MEMPART2 = 0000, 0000
MEMPART3 = 0000, 0000
MEMPART4 = 0000, 0000
MEMPART5 = 0000, 0000
MEMPART6 = 0000, 0000
MEMPART7 = 0000, 0000
PARMDRVA = 04, 02, Y
PARMDRVB = 04, 02, Y
PARMDRVC = 04, 02, Y
PARMDRVD = 04, 02, Y
PARMDRVE = 04, 02, Y
PARMDRVF = 04, 02, Y
PARMDRVG = 04, 02, Y
PARMDRVH = 04, 02, Y
PARMDRVI = 04, 02, Y
PARMDRVJ = 04, 02, Y
PARMDRVK = 04, 02, Y
PARMDRVL = 04, 02, Y
PARMDRVM = 04, 02, Y
PARMDRVN = 04, 02, Y
PARMDRVO = 04, 02, Y
PARMDRVP = 04, 02, Y
```

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_HSHTBL fields of the DPH.

EOF

CPMPIGA.WS4 (= "CP/M-86 Plus Installation Guide", appendix A)

(Edited by Emmanuel ROCHE.)

Appendix A: BIOS Development Method

This appendix presents an approach for implementing a CP/M-86 Plus BIOS in which the BIOS is increasingly refined in a series of steps. The purpose of this approach is to reduce the complexity of implementation and debugging during each step of the customization process. Each step consists of coding a section of the BIOS, creating the CPMP.SYS file with GENCPM, then debugging the new part of the BIOS under CP/M-86 1.X. Once this new part of the BIOS works, embark upon implementing the next step.

As you implement these steps, you might need to read or skim topics explained in the main body of this guide. For instance, as you work on the first step, you might need to read or review BIOS Kernel Data Header fields in Section 3, and Character Device Block fields in Section 6.

Use this BIOS development approach as simply a guideline, because experience varies, as well as the availability of hardware drivers already implemented for CP/M-86 1.X, Concurrent CP/M, or other operating systems. For example, if you already have Multisector disk I/O routines working, you can choose to skip the implementation of single sector I/O. Or, if you already have interrupt-driven console I/O routines, you can skip implementing polling character I/O drivers.

Table A-1. BIOS Development Method Steps

Step	Explanation
1	Implement simple input, input status, and output routines for the system console.
	a. Put the minimal amount of initialization code in the INIT module to allow console I/O to occur.
	b. Write code for console input, input status, and output in the CHARIO module. Define Character Device Block A (@CDBA) in the CHARIO module with the offsets of these drivers. Section 6 describes the CDB.
	c. Set the ?CLOCK_INIT routine to simply return. Do not enable a counter timer interrupt or a real-time clock interrupt.
	d. Assemble and link the BIOSKRNL, INIT, CHARIO, and CLOCK modules, as outlined in Section 9. The DISKIO module is excluded at this point, forcing all of the DPH symbols in the Kernel Data Header to a zero value. The MODEDIT utility is not needed for this development procedure. Use GENCPM to create the CPMP.SYS file. Answer the GENCPM question "Ticks per second ?" with a zero.

e. As described in Section 10, debug the character I/O routines. If you have not made the CCP resident, CP/M-86 Plus attempts to log in the initial default drive to read the CCP.CMD file. As none of the DPHs are defined, the Kernel IO_SELDSK (select disk) function returns an error, then the BDOS prints error messages to the console, and the Error CCP prompts with the following:

A>Cannot Load CCP

The Error CCP is built into the operating system, and is discussed in the "User's Guide". The Error CCP allows you only to change user numbers at this point. When you add the disk drivers, the default drive can be changed, and transients loaded through the Error CCP.

If you make the CCP a part of the memory image, the CCP attempts to read the STARTUP.SUB file, and the BDOS prints appropriate error messages. Then, the CCP displays the usual prompt. The BDOS also prints error messages on the logical CONOUT: device each time you attempt to load a transient when running the CCP or the Error CCP.

Use the command line editing functions to further test your console I/O routines (see C_READBUF in the "Programmer's Guide").

2 Add simple disk read routine.

To the DISKIO module, add a driver that reads physical sectors one at a time from drive A:. Specify A: to GENCPM as the initial default drive. Use the information in Section 8, and the Disk Parameter Block worksheet in Appendix D, to help you define a DPH named "@DPHA" and a DPB in the DISKIO module. Ensure @DPHA is declared a public symbol in your DISKIO module. LINK-86 places the offset of @DPHA that you define in the DISKIO module into the Kernel Data Header DPH table.

Provide the initialization code for the general hardware support of the disk in the INIT module. For instance, a DMA controller may need to be initialized. Set the DPH fields DPH_INIT, DPH_LOGIN, DPH_READ, and DPH_WRITE in the DISKIO module to the drive init, drive login, drive read, and drive write routines, which are also in your DISKIO module. For this step, set the drive login routine to simply return, and the drive write routine to return an error.

Translate multisector requests into single sector requests. Use simple CPU software loops for polling the disk controller, and use no interrupts. If necessary, reinitialize the PIC (Programmable Interrupt Controller) or similar hardware to disable interrupts from the disk controller.

After debugging, you should be able to perform DIR (directory) commands and load transients, but not write to the disk.

3 Add the disk write routine.

Use the same procedure as you used for disk read in step 2. Begin using scratch diskettes, if you are not already doing so, during debugging and testing.

4 Make console I/O routines interrupt-driven with type-ahead on input.

Section 4 describes interrupt devices in CP/M-86 Plus; Section 6 discusses interrupt character I/O in detail. Section 6 also supplies example routines, and the example BIOS for the CompuPro shows these routines in a working CHARIO module.

5 Make the disk I/O routines interrupt-driven. (Section 7)

6 Add other disk drivers and character drivers.

Consider implementing each device polled, then making it interrupt-driven, depending on how complex you expect the debugging task to be.

It is much easier to implement and test all the drivers required for your system before complicating matters with the tick interrupt in the CLOCK module.

7 Implement automatic density select and door open interrupt on the drives that support it. (Section 7)

8 Implement the tick interrupt, and test the system with background tasks. (Section 8)

EOF

CPMPIGB.WS4 (= "CP/M-86 Plus Installation Guide", appendix B)

(Edited by Emmanuel ROCHE.)

Appendix B: BIOS Kernel Listing

The BIOS Kernel is reproduced here for reference while reading the System Guide. The Kernel is also on the distribution disks in the file BIOSKRNL.A86. This BIOS Kernel listing includes the files SYSDAT.LIB, CDB.LIB, and DISK.LIB. A cross reference is included at the end of the listing.

Listing B-1. CP/M-86 Plus BIOS Kernel

```
 1
 2                       title   'CP/M-86 Plus BIOS Kernel'
 3                     ;*****************************************************
 4                     ;       Last Modification:      10/11/83
 5                     ;
 6                     ;               B I O S  -  8 6
 7                     ;               ===============
 8                     ;
 9                     ;       CP/M-86 PLUS Basic I/O System Kernel
10                     ;
11                     ;*****************************************************
12                     ;
13                     ;       Generation of BIOS3.SYS file
14                     ;
15                     ;       RASM86 bioskrnl
16                     ;       RASM86 init
17                     ;       RASM86 chario
18                     ;       RASM86 fdiskio
19                     ;       RASM86 clock
20                     ;
21                     ;       LINK86 bios3.sys = bioskrnl,init,chario,fdiskio,clock,zero.l86
22                     ;                       [search, data[origin[0F00]]]
23                     ;
24                     ;*****************************************************
25                     ;
26                     ;       Register usage for BIOS interface routines:
27                     ;
28                     ;       Entry:  AL = function number (in entry)
29                     ;               CX = first parameter
30                     ;               DX = second parameter
31                     ;               DS = system data segment (in entry and init)
32                     ;               ES = process environment (preserved through call)
33                     ;       Exit:   AX = return or BIOS error code
34                     ;               BX = AX (in exit)
35                     ;               DS = SYSDAT segment
36                     ;               ES = process environment (preserved through call)
37                     ;               SS,SP must also be preserved
38                     ;               CX,DX,SI,DI,BP can be changed by the BIOS
```

```
39                      ;
40                      ;*********************************************************
41                      ;
42                      ;       BIOS Kernel Coding Conventions
43                      ;
44                      ;*********************************************************
45                      ;
46                      ;       @ as the first character of a symbol denotes
47                      ;               a public variable
48                      ;
49                      ;       ? as the first character of a symbol denotes
50                      ;               a public label
51                      ;
52                      ;       All labels, and code are in lowercase
53                      ;               for easier reading.
54                      ;
55                      ;
56                      ;       All immediate values (literals) are in uppercase.
57                      ;
58                      ;       Fields within data structures have leading
59                      ;       letters followed by an underbar and the field
60                      ;       name.  The data structures defined are the following:
61                      ;
62                      ;               BH_     -       BIOS Header
63                      ;               CDB_    -       Character Device Block
64                      ;               DPB_    -       Disk Parameter Block
65                      ;               DPH_    -       Disk Parameter Header
66                      ;               IOPB_   -       I/O Parameter Block
67                      ;
68                      ;       Underscores are used for readiablity; otherwise,
69                      ;       symbols, code mnemonics, registers are in all uppercase
70                      ;       within comments to distinguish them from the code.
71                      ;
72                      ;       Each BIOS module has its publics and externals declared
73                      ;       all together within the code or data.
74                      ;
75                      ;*********************************************************
76
77 =                            include sysdat.lib
78 =                      ;*********************************************************
79 =                      ;
80 =                      ;       System Data Definitions
81 =                      ;
82 =                      ;*********************************************************
83 =
84 = 0030                bdos            equ     dword ptr .30h  ;entry into operating system
85 = 0034                int_dispatch    equ     dword ptr .34h  ;exit from interrupt handler
86 = 0038                int_setflag     equ     dword ptr .38h  ;interrupt SETFLAG function
87 = 003C                int_charscan    equ     dword ptr .3Ch  ;interrupt live key scanner
88 =
89 = 0046                osbaseseg       equ     word ptr .46h   ;base of the OS in para's
90 = 0048                osendseg        equ     word ptr .48h   ;first para after OS
91 =
92 = 004E                rlr             equ     word ptr .4Eh   ;ready list root
```

```
 93 =
 94 = 005F            tod_day       equ    word ptr .5Fh   ;number of days since 1/1/78
 95 = 0061            tod_hr        equ    byte ptr .61h   ;current hour in bcd
 96 = 0062            tod_min       equ    byte ptr .62h   ;current minute in bcd
 97 = 0063            tod_sec       equ    byte ptr .63h   ;current second in bcd
 98 =
 99 = 0066            con_width     equ    byte ptr .66h   ;console width
100 = 0067            con_len       equ    byte ptr .67h   ;console length
101 =
102 = 0046            err_mode      equ    es:byte ptr.46h ;process error mode in
103 =                                       ;in process descriptor
104
105
106 =                     include cdb.lib
107
108 =                 ;*******************************************************
109 =                 ;
110 =                 ;       Console Device Block Equates
111 =                 ;
112 =                 ;*******************************************************
113 =                 ;
114 =                 ;       +------+------+------+------+------+------+------+------+
115 =                 ; 00h: |              NAME               |   SUPCHAR   |
116 =                 ;       +------+------+------+------+------+------+------+------+
117 =                 ; 08h: |   CURCHAR   |SUPOEM|CUROEM| TXB  | RXB  | TYPE |IINPUT|
118 =                 ;       +------+------+------+------+------+------+------+------+
119 =                 ; 10h: |NFLAGS| RESRV|    COLINK   |    AOLINK   |    LOLINK   |
120 =                 ;       +------+------+-------------+-------------+-------------+
121 =                 ; 18h: |    INIT     |    INPUT    |   INSTAT    |   OUTPUT    |
122 =                 ;       +------+------+------+------+------+------+------+------+
123 =                 ; 20h: |   OUTSTAT   |
124 =                 ;       +------+------+
125 =
126 = 0000              CDB_NAME        equ     byte ptr 0
127 = 0006              CDB_SUPCHAR     equ     word ptr 6
128 = 0008              CDB_CURCHAR     equ     word ptr 8
129 = 000A              CDB_SUPOEM      equ     byte ptr 10
130 = 000B              CDB_CUROEM      equ     byte ptr 11
131 = 000C              CDB_TXB         equ     byte ptr 12
132 = 000D              CDB_RXB         equ     byte ptr 13
133 = 000E              CDB_TYPE        equ     byte ptr 14
134 = 000F              CDB_IINPUT      equ     byte ptr 15
135 = 0010              CDB_NFLAGS      equ     byte ptr 16
136 = 0011              CDB_RESRV       equ     byte ptr 17
137 = 0012              CDB_COLINK      equ     word ptr 18
138 = 0014              CDB_AOLINK      equ     word ptr 20
139 = 0016              CDB_LOLINK      equ     word ptr 22
140 = 0018              CDB_INIT        equ     word ptr 24
141 = 001A              CDB_INPUT       equ     word ptr 26
142 = 001C              CDB_INSTAT      equ     word ptr 28
143 = 001E              CDB_OUTPUT      equ     word ptr 30
144 = 0020              CDB_OUTSTAT     equ     word ptr 32
145 =
146 =                 ;       Equates for CDB_SUPCHAR fields
```

```
147 =
148 = 0001            CS_XON       equ    0001h
149 = 0002            CS_ETX       equ    0002h
150 = 0004            CS_RTS       equ    0004h
151 = 0008            CS_DTR       equ    0008h
152 = 0010            CS_POL       equ    0010h
153 = 0020            CS_ODD       equ    0020h
154 = 0040            CS_EVEN      equ    0040h
155 = 0080            CS_MARK      equ    0080h
156 = 0100            CS_SPACE     equ    0100h
157 = 0200            CS_5_DBITS   equ    0200h
158 = 0400            CS_6_DBITS   equ    0400h
159 = 0800            CS_7_DBITS   equ    0800h
160 =
161 = 1000            CS_8_DBITS   equ    1000h
162 = 2000            CS_1_SBITS   equ    2000h
163 = 4000            CS_15_SBITS  equ    4000h
164 = 8000            CS_2_SBITS   equ    8000h
165 =
166 =               ;     Equates for CDB_CURCHAR fields
167 =
168 = 0001            CC_XON       equ    0001h
169 = 0002            CC_ETX       equ    0002h
170 = 0004            CC_RTS       equ    0004h
171 = 0008            CC_DTR       equ    0008h
172 = 0010            CC_POL       equ    0010h
173 = 0020            CC_ENABLE    equ    0020h      ;enable parity
174 = 0000            CC_ODD       equ    0000h      ;  odd parity
175 = 0040            CC_EVEN      equ    0040h      ;  even parity
176 = 0080            CC_MARK      equ    0080h      ;  mark parity
177 = 00C0            CC_SPACE     equ    00C0h      ; space parity
178 = 0000            CC_5_DBITS   equ    0000h      ;5 data bits
179 = 0100            CC_6_DBITS   equ    0100h      ;6 data bits
180 = 0200            CC_7_DBITS   equ    0200h      ;7 data bits
181 = 0300            CC_8_DBITS   equ    0300h      ;8 data bits
182 = 0000            CC_1_SBITS   equ    0000h      ;  1 stop bit
183 = 0400            CC_15_SBITS  equ    0400h      ;1.5 stop bit
184 = 0800            CC_2_SBITS   equ    0800h      ;  2 stop bit
185 =
186 =               ;     Bit patterns to mask, and shift CDB_CURCHAR
187 =
188 = 00E0            CC_PARITY_MSK  equ   00E0h      ;parity enable and type mask
189 = 0005            CC_PARITY_SHF  equ   5          ;shift right for parity index
190 = 0300            CC_DBITS_MSK   equ   0300h      ;data bits mask
191 = 0008            CC_DBITS_SHF   equ   8          ;shift right for data bit index
192 = 0C00            CC_SBITS_MSK   equ   0C00h      ;stop bits mask
193 = 000A            CC_SBITS_SHF   equ   10         ;shift right for stop bit index
194 =
195 =               ;     Equates for CDB_TYPE field
196 =
197 = 0001            CT_INPUT     equ    01h        ;input  device
198 = 0002            CT_OUTPUT    equ    02h        ;output device
199 = 0004            CT_SOFTBAUD  equ    04h        ;software selectable baud
200 = 0008            CT_SERIAL    equ    08h        ;serial device
```

```
201 =
202 =                ;      Equates for CDB_BAUD field
203 =
204 = 0000             BAUD_NONE      equ     00h          ;  No  baud
205 = 0001             BAUD_50        equ     01h          ;  50  baud
206 = 0002             BAUD_625       equ     02h          ; 62.5  baud
207 = 0003             BAUD_75        equ     03h          ;  75  baud
208 = 0004             BAUD_110       equ     04h          ;  110  baud
209 = 0005             BAUD_1345      equ     05h          ;134.5  baud
210 = 0006             BAUD_150       equ     06h          ;  150  baud
211 = 0007             BAUD_200       equ     07h          ;  200  baud
212 = 0008             BAUD_300       equ     08h          ;  300  baud
213 =
214 = 0009             BAUD_600       equ     09h          ;  600  baud
215 = 000A             BAUD_1200      equ     0Ah          ; 1200  baud
216 = 000B             BAUD_1800      equ     0Bh          ; 1800  baud
217 = 000C             BAUD_2000      equ     0Ch          ; 2000  baud
218 = 000D             BAUD_2400      equ     0Dh          ; 2400  baud
219 = 000E             BAUD_3600      equ     0Eh          ; 3600  baud
220 = 000F             BAUD_4800      equ     0Fh          ; 4800  baud
221 = 0010             BAUD_7200      equ     10h          ; 7200  baud
222 = 0011             BAUD_9600      equ     11h          ; 9600  baud
223 = 0012             BAUD_192       equ     12h          ;19200  baud
224 = 0013             BAUD_384       equ     13h          ;38400  baud
225 = 0014             BAUD_56        equ     14h          ;56000  baud
226 = 0015             BAUD_768       equ     15h          ;76800  baud
227 = 0016             BAUD_OEM1      equ     16h
228 = 0017             BAUD_OEM2      equ     17h
229 = 0018             BAUD_OEM3      equ     18h
230
231
232 =                      include disk.lib
233 =                ;*******************************************************
234 =                ;
235 =                ;      Disk Parameter Header Equates
236 =                ;
237 =                ;*******************************************************
238 =                ;
239 =                ;     +------+------+------+------+------+------+------+------+
240 =                ; 00h |    XLT    |              | DOPEN|          |
241 =                ;     +------+------+------+------+------+------+------+------+
242 =                ; 08h |    DPB    |    CSV    |    ALV    |   DIRBCB   |
243 =                ;     +------+------+------+------+------+------+------+------+
244 =                ; 10h |   DATBCB   |   HSHTBL   |    INIT    |   LOGIN   |
245 =                ;     +------+------+------+------+------+------+------+------+
246 =                ; 18h |    READ    |   WRITE    | UNIT | CHNNL|NFLAGS|
247 =                ;     +------+------+------+------+------+------+------+
248 =
249 = 0000             DPH_XLT        equ     word ptr 0
250 = 0005             DPH_DOPEN      equ     byte ptr 5
251 = 0008             DPH_DPB        equ     word ptr 8
252 = 000A             DPH_CSV        equ     word ptr 10
253 = 000C             DPH_ALV        equ     word ptr 12
254 = 000E             DPH_DIRBCB     equ     word ptr 14
```

```
255 = 0010            DPH_DATBCB      equ     word ptr 16
256 = 0012            DPH_HSHTBL      equ     word ptr 18
257 = 0014            DPH_INIT        equ     word ptr 20
258 = 0016            DPH_LOGIN       equ     word ptr 22
259 = 0018            DPH_READ        equ     word ptr 24
260 = 001A            DPH_WRITE       equ     word ptr 26
261 = 001C            DPH_UNIT        equ     byte ptr 28
262 = 001D            DPH_CHNNL       equ     byte ptr 29
263 = 001E            DPH_NFLAGS      equ     byte ptr 30
264 =
265 =               ;*****************************************************
266 =               ;
267 =               ;
268 =               ;       Disk Parameter Block Equates
269 =               ;
270 =               ;*****************************************************
271 =               ;
272 =               ;       +-----+-----+-----+-----+-----+-----+-----+-----+
273 =               ;  00h |    SPT    | BSH | BLM | EXM |    DSM    |  DRM..
274 =               ;       +-----+-----+-----+-----+-----+-----+-----+-----+
275 =               ;  08h  ..DRM | AL0 | AL1 |    CKS    |    OFF    | PSH |
276 =               ;       +-----+-----+-----+-----+-----+-----+-----+-----+
277 =               ;  10h  | PHM |
278 =               ;       +-----+
279 =
280 = 0000            DPB_SPT         equ     word ptr 0
281 = 0002            DPB_BSH         equ     byte ptr 2
282 = 0003            DPB_BLM         equ     byte ptr 3
283 = 0004            DPB_EXM         equ     byte ptr 4
284 = 0005            DPB_DSM         equ     word ptr 5
285 = 0007            DPB_DRM         equ     word ptr 7
286 = 0009            DPB_AL0         equ     byte ptr 9
287 = 000A            DPB_AL1         equ     byte ptr 10
288 = 000B            DPB_CKS         equ     word ptr 11
289 = 000D            DPB_OFF         equ     word ptr 13
290 = 000F            DPB_PSH         equ     byte ptr 15
291 = 0010            DPB_PHM         equ     byte ptr 16
292 =
293 =               ;*****************************************************
294 =               ;
295 =               ;       Input/Output Parameter Block Definition
296 =               ;
297 =               ;*****************************************************
298 =               ;
299 =               ;  Read and Write disk parameter equates
300 =               ;
301 =               ;  At the disk read and write entries,
302 =               ;   all disk I/O parameters are on the stack
303 =               ;   and the stack at these entries appears as
304 =               ;   follows:
305 =               ;
306 =               ;         +-------+-------+
307 =               ;       +14 |  DRV  | MCNT  |   Drive and Multisector count
308 =               ;         +-------+-------+
```

```
309 =                ;     +12 |    TRACK    |  Track number
310 =                ;       +-------+-------+
311 =                ;     +10 |    SECTOR   |  Physical sector number
312 =                ;       +-------+-------+
313 =                ;      +8 |    DMA_SEG  |  DMA segment
314 =                ;       +-------+-------+
315 =                ;      +6 |    DMA_OFF  |  DMA offset
316 =                ;       +-------+-------+
317 =                ;      +4 |    RET_SEG  |  BDOS return segment
318 =                ;       +-------+-------+
319 =                ;
320 =                ;      +2 |    RET_OFF  |  BDOS return offset
321 =                ;       +-------+-------+
322 =                ;    BP+0 |    RET_ADR  |  Local ENTRY return address
323 =                ;       +-------+-------+   (assumes one level of call
324 =                ;                           from ENTRY routine)
325 =                ;
326 =                ; These parameters may be indexed and modified
327 =                ;  directly on the stack and will be removed
328 =                ;  by the BDOS after the function is complete.
329 =
330 = 000F             iopb_mcnt      equ    byte ptr 15[bp]
331 = 000E             iopb_drive     equ    byte ptr 14[bp]
332 = 000C             iopb_track     equ    word ptr 12[bp]
333 = 000A             iopb_sector    equ    word ptr 10[bp]
334 = 0008             iopb_dmaseg    equ    word ptr 8[bp]
335 = 0006             iopb_dmaoff    equ    word ptr 6[bp]
336
337
338                  ;********************************************************
339                  ;
340                  ;     BIOS Kernel Code Publics and Externals
341                  ;
342                  ;********************************************************
343                       CSEG
344
345                       public  @sysdat              ;force CS override
346                       public  ?waitflag, ?delay, ?dispatch, ?pmsg
347
348                       extrn   ?init:near, ?clock_init:near
349
350                  ;********************************************************
351                  ;
352                  ;     BIOS Code Header
353                  ;
354                  ;********************************************************
355                       org     0000h
356
357  0000 E90500         0008       jmp biosinit              ;BIOS initialization entry
358  0003 E94000         0046       jmp biosentry              ;BIOS function entry
359
360  0006              @sysdat       rw     1           ;OS Data Segment
361
362                  ;********************************************************
```

```
363                 ;
364                 ;       BIOS Kernel Data Publics and Externals
365                 ;
366                 ;*********************************************************
367                       DSEG
368
369                       public  @bh_delay, @bh_ticksec, @bh_gdopen, @bh_inint
370                       public  @bh_nextflag, @bh_lastflag, @bh_intconin, @bh_8087
371                       public  @bh_dphtable, @bh_cdbtable
372
373                       public  @bh_ciroot, @bh_coroot, @bh_airoot, @bh_aoroot, @bh_loroot
374                       public  @bh_bufbase, @bh_buflen, @bh_memdesc
375
376                       extrn   @dpha:word, @dphb:word, @dphc:word, @dphd:word
377                       extrn   @dphe:word, @dphf:word, @dphg:word, @dphh:word
378                       extrn   @dphi:word, @dphj:word, @dphk:word, @dphl:word
379                       extrn   @dphm:word, @dphn:word, @dpho:word, @dphp:word
380
381                       extrn   @cdba:word, @cdbb:word, @cdbc:word, @cdbd:word
382                       extrn   @cdbe:word, @cdbf:word, @cdbg:word, @cdbh:word
383                       extrn   @cdbi:word, @cdbj:word, @cdbk:word, @cdbl:word
384                       extrn   @cdbm:word, @cdbn:word, @cdbo:word, @cdbp:word
385
386                       extrn   @signon:byte
387
388                 ;*********************************************************
389                 ;
390                 ;       BIOS Data Header
391                 ;
392                 ;*********************************************************
393                       org     0000h
394                           ;use the LINK-86 [data[origin[0F00]]] option
395                           ;to set the origin of the data segment at 0F00h
396
397   0000 00           @bh_delay       db      0               ;0FFh if process delaying
398   0001 3C           @bh_ticksec     db      60              ;ticks per second
399   0002 00           @bh_gdopen      db      0               ;0FFh if drive door opened
400   0003 00           @bh_inint       db      0               ;in interrupt count
401   0004 04           @bh_nextflag    db      4               ;next available flag
402   0005 00           @bh_lastflag    db      0               ;last available flag
403   0006 00           @bh_intconin    db      0               ;0FFh if interrupt driven CONIN:
404   0007 00           @bh_8087        db      0               ;0FFh if 8087 exists
405
406                 ;       disk parameter header offset table
407
408   0008 0000     E   @bh_dphtable   dw      offset @dpha    ;drive A:
409   000A 0000     E                   dw      offset @dphb    ;drive B:
410   000C 0000     E                   dw      offset @dphc    ;drive C:
411   000E 0000     E                   dw      offset @dphd    ;drive D:
412   0010 0000     E                   dw      offset @dphe    ;drive E:
413   0012 0000     E                   dw      offset @dphf    ;drive F:
414   0014 0000     E                   dw      offset @dphg    ;drive G:
415   0016 0000     E                   dw      offset @dphh    ;drive H:
416   0018 0000     E                   dw      offset @dphi    ;drive I:
```

```
417  001A 0000       E                dw      offset @dphj    ;drive J:
418  001C 0000       E                dw      offset @dphk    ;drive K:
419  001E 0000       E                dw      offset @dphl    ;drive L:
420  0020 0000       E                dw      offset @dphm    ;drive M:
421  0022 0000       E                dw      offset @dphn    ;drive N:
422  0024 0000       E                dw      offset @dpho    ;drive O:
423  0026 0000       E                dw      offset @dphp    ;drive P:
424
425
426                  ;       character device block offset table
427
428  0028 0000       E    @bh_cdbtable    dw      offset @cdba    ;device A
429  002A 0000       E                dw      offset @cdbb    ;device B
430  002C 0000       E                dw      offset @cdbc    ;device C
431  002E 0000       E                dw      offset @cdbd    ;device D
432  0030 0000       E                dw      offset @cdbe    ;device E
433  0032 0000       E                dw      offset @cdbf    ;device F
434  0034 0000       E                dw      offset @cdbg    ;device G
435  0036 0000       E                dw      offset @cdbh    ;device H
436  0038 0000       E                dw      offset @cdbi    ;device I
437  003A 0000       E                dw      offset @cdbj    ;device J
438  003C 0000       E                dw      offset @cdbk    ;device K
439  003E 0000       E                dw      offset @cdbl    ;device L
440  0040 0000       E                dw      offset @cdbm    ;device M
441  0042 0000       E                dw      offset @cdbn    ;device N
442  0044 0000       E                dw      offset @cdbo    ;device O
443  0046 0000       E                dw      offset @cdbp    ;device P
444
445                  ;       Character device roots for console input,
446                  ;       console ouput, auxiliary input, auxiliary output
447                  ;       and list output.
448
449  0048 0000       E    @bh_ciroot      dw      offset @cdba    ;console input
450  004A 0000       E    @bh_coroot      dw      offset @cdba    ;console output
451  004C 0000       E    @bh_airoot      dw      offset @cdbb    ;aux input
452  004E 0000       E    @bh_aoroot      dw      offset @cdbb    ;aux output
453  0050 0000       E    @bh_loroot      dw      offset @cdbc    ;list output
454
455  0052 0000            @bh_bufbase     dw      0               ;offset of buffer
456  0054 0000            @bh_buflen      dw      0               ;length of buffer
457
458  0056               @bh_memdesc     rw      32*3            ;room for 32 memory descriptors
459
460                  ;       O.S. error messages
461
462  0116 5401       R    bh_chain        dw      chain_msg       ;chain error message address
463  0118 6C01       R    bh_prompt       dw      prompt_msg      ;error CCP prompt message address
464  011A 7F01       R    bh_user         dw      user_str        ;error CCP command string
465  011C 8401       R    bh_cpmerr       dw      cmperr_msg      ;CP/M error message address
466  011E 9501       R    bh_func         dw      func_msg        ;function message address
467  0120 A801       R    bh_file         dw      file_msg        ;file message address
468  0122 B101       R    bh_err1         dw      err1_msg
469  0124 C701       R    bh_err2         dw      err2_msg
470  0126 D601       R    bh_err3         dw      err3_msg
```

```
0128 E501       R     bh_err4        dw      err4_msg
012A F301       R     bh_err5        dw      err5_msg
012C 0202       R     bh_err6        dw      err6_msg
012E 0E02       R     bh_err7        dw      err7_msg

                ;*********************************************************
                ;
                ;
                ;       BIOS Code Segment
                ;
                ;*********************************************************
                        CSEG

                ;========
                biosinit:
                ;========
                ;       Entry:  DS = system data segment
                ;               ES = process environment (preserved through call)
                ;       Exit:   DS = SYSDAT segment
                ;               ES = process environment (preserved through call)

0008 E80000     E               call ?init                      ;perform any general initialization

000B 33F6                       xor si,si                       ;index into tables
000D B91000                     mov cx,16                       ;16 total drives and devices
                next_device:
0010 51                         push cx
0011 8B9C2800   R               mov bx,@bh_cdbtable[si]         ;offset of CDB in table
0015 0BDB                       or bx,bx                        ;is offset zero
0017 7407       0020            jz init_drv                     ;yes no character device
0019 32D2                       xor dl,dl                       ;DL = 0, first call to CDB init
001B 56                         push si
001C FF5718                     call CDB_INIT[bx]
001F 5E                         pop si
                init_drv:
0020 8B9C0800   R               mov bx,@bh_dphtable[si]         ;offset of DPH in table
0024 0BDB                       or bx,bx                        ;is it zero
0026 7405       002D            jz zero_entry                   ;yes get next device
0028 56                         push si
0029 FF5714                     call DPH_INIT[bx]
002C 5E                         pop si
                zero_entry:
002D 4646                       inc si ! inc si
002F 59                         pop cx
0030 E2DE       0010            loop next_device

0032 8B1E4800   R               mov bx,@bh_ciroot
0036 8A470F                     mov al,CDB_IINPUT[bx]
0039 A20600     R               mov @bh_intconin,al             ;setup console input interrupt flag

003C E80000     E               call ?clock_init                ;initialize the system clock

003F BE0000     E               mov si,offset @signon           ;sign-on message defined in INIT module
0042 E8FE00     0143            call ?pmsg                      ;print the BIOS sign-on message
```

```
525  0045 CB                    retf
526
527                     ;*******************************************************
528                     ;
529                     ;       BIOS Entry Function Dispatch
530                     ;
531                     ;
532                     ;*******************************************************
533
534                     ;=========
535                     biosentry:      ; BIOS Entry Point
536                     ;=========
537                     ; All calls to the BIOS after INIT, enter through this code
538                     ;  with a CALLF and must return with a RETF.
539                     ;       Entry:  AL = function number
540                     ;               CX = first parameter
541                     ;               DX = second parameter
542                     ;               DS = system data segment
543                     ;               ES = process environment (preserved through call)
544                     ;       Exit:   AX = BX = return or BIOS error code
545                     ;               DS = SYSDAT segment
546                     ;               ES = process environment (preserved through call)
547                     ;               SS,SP must also be preserved
548                     ;               CX,DX,SI,DI,BP can be changed by the BIOS
549
550  0046 3C807312      005C        cmp al,80h ! jae range_err      ;check for BIOS functions above 80h
551  004A FC                    cld                         ;clear direction flag
552  004B 32E4D1E0                  xor ah,ah ! shl ax,1         ;index into BIOS function table
553  004F 8BD8                  mov bx,ax
554  0051 FF973001     R            call functab[bx]            ;call BIOS kernel routine
555  0055 8E064E00                  mov es,rlr                ;restore ES
556                     bdos_ret:
557  0059 8BD8                  mov bx,ax                ;BX = AX
558  005B CB                    retf
559                     range_err:
560  005C B8FFFF                mov ax,0FFFFh                ;function out of range
561  005F EBF8          0059        jmps bdos_ret
562
563                     ;*******************************************************
564                     ;
565                     ;       BIOS Device Initialization Routines
566                     ;
567                     ;*******************************************************
568
569                     io_devinit:     ;BIOS function 14
570                     ;==========
571                     ;       Entry:  CX = Offset of CDB
572                     ;       Exit:   AX = 0 if successful
573                     ;               = 0FFFFh if error
574
575  0061 8BD9                  mov bx,cx
576  0063 B201                  mov dl,1                 ;DL = 1, not first call to CDB init
577  0065 FF6718                jmp CDB_INIT[bx]
578
```

```
                ;*********************************************************
                ;
                ;       BIOS Character Input Status Routines
                ;
                ;*********************************************************


                io_conist:      ;BIOS function 0
                ;=========
                ;       Entry:  None
                ;       Exit:   AL = 0FFH if ready
                ;               AL = 000H if not ready

0068 8B1E4800   R               mov bx,@bh_ciroot               ;console input root
                ;       jmps ist_scan                   ;fall through to IST_SCAN

                ist_scan:
                ;--------
006C 0BDB7403           0073            or bx,bx ! jz no_stat_dev
0070 FF671C                             jmp CDB_INSTAT[bx]
                no_stat_dev:
0073 33C0                               xor ax,ax
0075 C3                                 ret

                io_auxist:      ;BIOS function 16
                ;=========
                ;       Entry:  None
                ;       Exit:   AL = 0FFH if ready
                ;               AL = 000H if not ready

0076 8B1E4C00   R               mov bx,@bh_airoot               ;aux input root
007A EBF0               006C            jmps ist_scan

                ;*********************************************************
                ;
                ;       BIOS Character Output Status Routines
                ;
                ;*********************************************************

                io_liststst:    ;BIOS function 3
                ;=========
                ;       Entry:  None
                ;       Exit:   AL = 0FFH if ready
                ;               AL = 000H if not ready

007C 8B1E5000   R               mov bx,@bh_loroot               ;list root
0080 BF1600                             mov di,CDB_LOLINK
0083 EB07               008C            jmps ost_scan

                io_conost:      ;BIOS function 15
                ;=========
                ;       Entry:  None
                ;       Exit:   AL = 0FFH if ready
                ;               AL = 000H if not ready
```

```
633
634 0085 8B1E4A00    R           mov bx,@bh_coroot           ;console output root
635 0089 BF1200                  mov di,CDB_COLINK
636                 ;     jmps ost_scan               ;fall through to OST_SCAN
637
638
639                 ost_scan:     ;output status scanner
640                 ;--------
641                 ;     Entry:  BX = offset of CDB
642                 ;             DI = offset within CDB to next link
643                 ;     Exit:   AL = 0FFH if ready
644                 ;             AL = 000H if not ready
645
646 008C 0BDB740F     009F       or bx,bx ! jz ost_rdy       ;if zero status's defined ready
647 0090 5357                    push bx ! push di
648 0092 FF5720                   call CDB_OUTSTAT[bx]        ;perform output status check
649 0095 5F5B                     pop di ! pop bx
650 0097 0AC07406     00A1        or al,al ! jz ost_notrdy    ;if one device not ready,none are.
651
652 009B 8B19                     mov bx,[di+bx]
653 009D EBED         008C        jmps ost_scan
654                 ost_rdy:
655 009F B0FF                     mov al,0FFh
656                 ost_notrdy:
657 00A1 C3                       ret
658
659                 io_auxost:    ;BIOS function 17
660                 ;=========
661                 ;     Entry:  None
662                 ;     Exit:   AL = 0FFH if ready
663                 ;             AL = 000H if not ready
664
665 00A2 8B1E4E00    R           mov bx,@bh_aoroot           ;aux output root
666 00A6 BF1400                   mov di,CDB_AOLINK
667 00A9 EBE1         008C        jmps ost_scan
668
669                 ;*******************************************************
670                 ;
671                 ;     BIOS Character Input Routines
672                 ;
673                 ;*******************************************************
674
675                 io_conin:     ;BIOS function 1
676                 ;========
677                 ;     Entry:  None
678                 ;     Exit:   AL = character
679
680 00AB 8B1E4800    R           mov bx,@bh_ciroot           ;console input root
681                 ;     jmps in_scan                ;fall through to IN_SCAN
682
683                 in_scan:      ;character output
684                 ;-------
685                 ;     Entry:  BX = offset of CDB
686                 ;     Exit:   AL = character
```

```
687
688 00AF 0BDB7403      00B6        or bx,bx ! jz no_input_dev
689 00B3 FF671A                 jmp CDB_INPUT[bx]           ;get character
690
691                 no_input_dev:
692 00B6 33C0                    xor ax,ax
693 00B8 C3                     ret
694
695                 io_auxin:       ;BIOS function 5
696                 ;========
697                 ;       Entry:  None
698                 ;       Exit:   AL = character
699
700 00B9 8B1E4C00    R           mov bx,@bh_airoot           ;aux input root
701 00BD EBF0           00AF        jmps in_scan                ;get character
702
703                 ;*******************************************************
704                 ;
705                 ;       BIOS Character Output Routines
706                 ;
707                 ;*******************************************************
708
709                 io_conout:      ;BIOS function 2
710                 ;=========
711                 ;       Entry:  CL = character
712                 ;       Exit:   None
713
714 00BF 8B1E4A00    R           mov bx,@bh_coroot           ;console output root
715 00C3 BF1200                 mov di,CDB_COLINK
716                 ;       jmps out_scan               ;fall through to OUT_SCAN
717
718                 out_scan:       ;character output
719                 ;--------
720                 ;       Entry:  CL = character
721                 ;               BX = offset of CDB
722                 ;               DI = offset within CDB to next link
723                 ;       Exit:   None
724
725 00C6 0BDB740D      00D7        or bx,bx ! jz out_exit          ;zero = done
726 00CA 535157                 push bx ! push cx ! push di   ;save the character
727 00CD FF571E                  call CDB_OUTPUT[bx]         ;output the character
728 00D0 5F595B                  pop di ! pop cx ! pop bx
729 00D3 8B19                   mov bx,[di+bx]              ;get next cdb offset
730 00D5 EBEF           00C6        jmps out_scan
731                 out_exit:
732 00D7 C3                     ret
733
734                 io_list:        ;BIOS function 4
735                 ;=======
736                 ;       Entry:  CL = character
737                 ;       Exit:   None
738
739 00D8 8B1E5000    R           mov bx,@bh_loroot           ;list output root
740 00DC BF1600                  mov di,CDB_LOLINK
```

```
00DF EBE5         00C6       jmps out_scan


                  io_auxout:     ;BIOS function 6
                  ;=========
                  ;      Entry:  CL = character
                  ;      Exit:   None

00E1 8B1E4E00     R            mov bx,@bh_aoroot             ;aux output root
00E5 BF1400                    mov di,CDB_AOLINK
00E8 EBDC         00C6         jmps out_scan

                  ;*******************************************************
                  ;
                  ;      BIOS Disk I/O Routines
                  ;
                  ;*******************************************************

                  io_seldsk:     ;BIOS function 9
                  ;=========
                  ;      Entry:  CL = disk to be selected
                  ;              DL = (Bit 0): 0 if first select
                  ;      Exit:   AX = 0 if illegal select
                  ;              = offset of DPH relative to OS data segment

00EA 33DB                      xor bx,bx
00EC 80F90F7714     0105       cmp cl,15 ! ja sel_ret         ;if not valid drive exit
00F1 8AD9D1E3                  mov bl,cl ! shl bx,1         ;double drive number
00F5 8B9F0800     R            mov bx,@bh_dphtable[bx]      ;index into drive table
00F9 0BDB7408       0105         or bx,bx ! jz sel_ret        ;zero = bad select
00FD F6C2017503     0105           test dl,1 ! jnz sel_ret     ;first time select?
0102 FF5716                    call DPH_LOGIN[bx]       ; yes
                  sel_ret:
0105 8BC3                    mov ax,bx
0107 C3                    ret

                  io_read:       ;BIOS function 10
                  ;=======
                  ;      Entry:  IOPB filled in (on stack)
                  ;      Exit:   AL =    0 if no error
                  ;              =    1 if physical error
                  ;              = 0FFH if media density has changed

0108 BF1800                    mov di,DPH_READ              ;DPH read routine offset
010B EB05         0112         jmps read_write              ;jump to common i/o routine

                  io_write:      ;BIOS function 11
                  ;========
                  ;      Entry:  IOPB filled in (onstack)
                  ;      Exit:   AL =    0 if no error
                  ;              =    1 if physical error
                  ;              =    2 if Read/Only disk
                  ;              = 0FFH if media density has changed

```

```
795  010D BF1A00                mov di,DPH_WRITE            ;DPH write routine offset
796
797  0110 EB00       0112       jmps read_write             ;jump to common i/o routine
798
799                 ;*******************************************************
800                 ;
801                 ;       BIOS Disk I/O Common Read/Write Routines
802                 ;
803                 ;*******************************************************
804
805                 read_write:     ;checks for valid disk and calls read or write
806                 ;----------     ; routine for that drive
807                 ;
808                 ;       Entry:  DI = offset of read or write routine in DPH
809                 ;       Exit:   AX = return code
810
811  0112 8BEC                  mov bp,sp                   ;SS:BP points to IOPB
812  0114 8A5E0E                 mov bl,iopb_drive
813  0117 32FFD1E3                xor bh,bh ! shl bx,1
814  011B 8B9F0800    R           mov bx,@bh_dphtable[bx]        ;get DPH address
815  011F 0BDB7402     0125       or bx,bx! jz ret_error         ;check if valid
816  0123 FF21                   jmp word ptr [bx+di]         ;jump to DPH read or write routine
817                 ret_error:
818  0125 B001                   mov al,1                    ;return error if not valid
819  0127 C3                    ret
820
821                 io_flush:       ;BIOS function 12
822                 ;========
823                 ;       Entry:  None
824                 ;       Exit:   AL = 0 if no error
825                 ;               = 1 if physical error
826                 ;               = 2 if Read/Only disk
827
828  0128 33C0                   xor ax,ax                   ;flush not necessary
829  012A C3                    ret                         ;when BDOS deblocking
830
831                 ;*******************************************************
832                 ;
833                 ;       Concurrent-86 Functions Not Implemented
834                 ;
835                 ;*******************************************************
836
837                 io_notimp:      ;BIOS function 7,8,13
838                 ;=========
839  012B 33C0                   xor ax,ax                   ;return success
840  012D C3                    ret
841
842                 ;*******************************************************
843                 ;
844                 ;       Public Routine Code Segment
845                 ;
846                 ;*******************************************************
847
848                 ?waitflag:      ;FLAG WAIT  -  Wait for pseudo interrupt
```

```
849
850                     ;=========
851                     ;       Entry:  DL = number of flag to wait on.
852                     ;       Exit:   AL = 0
853
854  012E B184                  mov cl,132              ;flagwait function number
855  0130 FF1E3000                 callf bdos              ;call the OS
856  0134 C3                 ret                     ;return to BIOS caller
857
858                     ?delay:         ;DELAY - Delay specified no. of system ticks
859                     ;======
860                     ;       Entry:  DX = number of system ticks to delay
861                     ;       Exit:   None
862
863  0135 B18D                  mov cl,141              ;delay function number
864  0137 FF1E3000                 callf bdos              ;call the OS
865  013B C3                 ret                     ;return to BIOS caller
866
867                     ?dispatch:      ;DISPATCH - force a dispatch
868                     ;=========
869                     ;       Entry:  None
870                     ;       Exit:   None
871
872  013C B18E                  mov cl,142              ;dispatch function number
873  013E FF1E3000                 callf bdos              ;call the OS
874  0142 C3                 ret                     ;return to BIOS caller
875
876                     ?pmsg:          ;Print String
877                     ;=====
878                     ;       Entry:  DS:SI = offset of string terminated by 0.
879                     ;               ES = process environment segment
880                     ;       Exit:   None
881
882  0143 26803E4600FF             cmp err_mode,0FFh             ;if return error mode is set
883  0149 740E          0159       je pmsg_exit                ; dont print message
884                     next_char:
885  014B 8A0C                  mov cl,[si]             ;get character from buffer
886  014D 0AC97408      0159        or cl,cl ! jz pmsg_exit      ;check for 0 terminater
887  0151 56                    push si
888  0152 E86AFF        00BF            call io_conout          ;output character in CL
889  0155 5E                    pop si
890  0156 46                    inc si
891  0157 EBF2          014B            jmps next_char          ;get next character
892                     pmsg_exit:
893  0159 C3                 ret
894
895                     ;*******************************************************
896                     ;
897                     ;       BIOS Data Segment
898                     ;
899                     ;*******************************************************
900                         DSEG
901
902
```

```
903                     ;       BIOS Function Table
904
905  0130 6800        R     functab         dw      io_conist       ; 0 - console status
906  0132 AB00         R                    dw      io_conin        ; 1 - console input
907  0134 BF00         R                    dw      io_conout       ; 2 - console output
908  0136 7C00         R                    dw      io_liststst     ; 3 - list output status
909  0138 D800         R                    dw      io_list         ; 4 - list output
910  013A B900         R                    dw      io_auxin        ; 5 - aux input
911  013C E100         R                    dw      io_auxout       ; 6 - aux output
912  013E 2B01         R                    dw      io_notimp       ; 7 - CCP/M function
913  0140 2B01         R                    dw      io_notimp       ; 8 - CCP/M function
914  0142 EA00         R                    dw      io_seldsk       ; 9 - select disk
915  0144 0801         R                    dw      io_read         ;10 - read sector
916  0146 0D01         R                    dw      io_write        ;11 - write sector
917  0148 2801         R                    dw      io_flush        ;12 - flush buffers
918  014A 2B01         R                    dw      io_notimp       ;13 - CCP/M function
919  014C 6100         R                    dw      io_devinit      ;14 - char. device init
920  014E 8500         R                    dw      io_conost       ;15 - console output status
921  0150 7600         R                    dw      io_auxist       ;16 - aux input status
922  0152 A200         R                    dw      io_auxost       ;17 - aux output status
923
924  0154 0D0A43616E6E       chain_msg       db      13,10,'Cannot Load Program',13,10,'$'
925       6F74204C6F61
926       642050726F67
927       72616D0D0A24
928  016C 0D0A43616E6E       prompt_msg      db      13,10,'Cannot Load CCP $'
929       6F74204C6F61
930       642043435020
931       24
932  017F 0455534552         user_str        db      4,'USER'
933  0184 0D0A43502F4D       cmperr_msg      db      13,10,'CP/M Error On $'
934       204572726F72
935       204F6E2024
936  0195 0D0A42444F53       func_msg        db      13,10,'BDOS Function = $'
937       2046756E6374
938       696F6E203D20
939       24
940  01A8 2046696C6520       file_msg        db      ' File = $'
941       3D2024
942  01B1 4469736B2052       err1_msg        db      'Disk Read/Write Error$'
943       6561642F5772
944       697465204572
945       726F7224
946  01C7 526561642D4F       err2_msg        db      'Read-Only Disk$'
947       6E6C79204469
948       736B24
949  01D6 526561642D4F       err3_msg        db      'Read-Only File$'
950       6E6C79204669
951       6C6524
952  01E5 496E76616C69       err4_msg        db      'Invalid Drive$'
953       642044726976
954       6524
955
956  01F3 50617373776F       err5_msg        db      'Password Error$'
```

```
  957       726420457272
  958       6F7224
  959   0202 46696C652045       err6_msg        db      'File Exists$'
  960       786973747324
  961   020E 3F20696E2046       err7_msg        db      '? in Filename$'
  962       696C656E616D
  963       6524
  964
  965                           END
  966
  967
  968  END OF ASSEMBLY.  NUMBER OF ERRORS:   0.  USE FACTOR: 10%

?CLOCK_INIT      0000 L   348   521
?DELAY           0135 L   346   858#
?DISPATCH        013C L   346   867#
?INIT            0000 L   348   492
?PMSG            0143 L   346   524   876#
?WAITFLAG        012E L   346   848#
@BH_8087         0007 V   370   404#
@BH_AIROOT       004C V   373   451#  609   700
@BH_AOROOT       004E V   373   452#  665   749
@BH_BUFBASE      0052 V   374   455#
@BH_BUFLEN       0054 V   374   456#
@BH_CDBTABLE     0028 V   371   428#  498
@BH_CIROOT       0048 V   373   449#  517   592   680
@BH_COROOT       004A V   373   450#  634   714
@BH_DELAY        0000 V   369   397#
@BH_DPHTABLE     0008 V   371   408#  506   769   814
@BH_GDOPEN       0002 V   369   399#
@BH_ININT        0003 V   369   400#
@BH_INTCONIN     0006 V   370   403#  519
@BH_LASTFLAG     0005 V   370   402#
@BH_LOROOT       0050 V   373   453#  624   739
@BH_MEMDESC      0056 V   374   458#
@BH_NEXTFLAG     0004 V   370   401#
@BH_TICKSEC      0001 V   369   398#
@CDBA            0000 V   381   428   449   450
@CDBB            0000 V   381   429   451   452
@CDBC            0000 V   381   430   453
@CDBD            0000 V   381   431
@CDBE            0000 V   382   432
@CDBF            0000 V   382   433
@CDBG            0000 V   382   434
@CDBH            0000 V   382   435
@CDBI            0000 V   383   436
@CDBJ            0000 V   383   437
@CDBK            0000 V   383   438
@CDBL            0000 V   383   439
@CDBM            0000 V   384   440
@CDBN            0000 V   384   441
@CDBO            0000 V   384   442
@CDBP            0000 V   384   443
@DPHA            0000 V   376   408
```

@DPHB 0000 V 376 409
@DPHC 0000 V 376 410
@DPHD 0000 V 376 411
@DPHE 0000 V 377 412
@DPHF 0000 V 377 413
@DPHG 0000 V 377 414
@DPHH 0000 V 377 415
@DPHI 0000 V 378 416
@DPHJ 0000 V 378 417
@DPHK 0000 V 378 418
@DPHL 0000 V 378 419
@DPHM 0000 V 379 420
@DPHN 0000 V 379 421
@DPHO 0000 V 379 422
@DPHP 0000 V 379 423
@SIGNON 0000 V 386 523
@SYSDAT 0006 V 345 360#
BAUD_110 0004 N 208#
BAUD_1200 000A N 215#
BAUD_1345 0005 N 209#
BAUD_150 0006 N 210#
BAUD_1800 000B N 216#
BAUD_192 0012 N 223#
BAUD_200 0007 N 211#
BAUD_2000 000C N 217#
BAUD_2400 000D N 218#
BAUD_300 0008 N 212#
BAUD_3600 000E N 219#
BAUD_384 0013 N 224#
BAUD_4800 000F N 220#
BAUD_50 0001 N 205#
BAUD_56 0014 N 225#
BAUD_600 0009 N 214#
BAUD_625 0002 N 206#
BAUD_7200 0010 N 221#
BAUD_75 0003 N 207#
BAUD_768 0015 N 226#
BAUD_9600 0011 N 222#
BAUD_NONE 0000 N 204#
BAUD_OEM1 0016 N 227#
BAUD_OEM2 0017 N 228#
BAUD_OEM3 0018 N 229#
BDOS 0030 V 84# 855 864 873
BDOS_RET 0059 L 556# 561
BH_CHAIN 0116 V 462#
BH_CPMERR 011C V 465#
BH_ERR1 0122 V 468#
BH_ERR2 0124 V 469#
BH_ERR3 0126 V 470#
BH_ERR4 0128 V 471#
BH_ERR5 012A V 472#
BH_ERR6 012C V 473#
BH_ERR7 012E V 474#
BH_FILE 0120 V 467#

BH_FUNC 011E V 466#
BH_PROMPT 0118 V 463#
BH_USER 011A V 464#
BIOSENTRY 0046 L 358 535#
BIOSINIT 0008 L 357 485#
CC_15_SBITS 0400 N 183#
CC_1_SBITS 0000 N 182#
CC_2_SBITS 0800 N 184#
CC_5_DBITS 0000 N 178#
CC_6_DBITS 0100 N 179#
CC_7_DBITS 0200 N 180#
CC_8_DBITS 0300 N 181#
CC_DBITS_MSK 0300 N 190#
CC_DBITS_SHF 0008 N 191#
CC_DTR 0008 N 171#
CC_ENABLE 0020 N 173#
CC_ETX 0002 N 169#
CC_EVEN 0040 N 175#
CC_MARK 0080 N 176#
CC_ODD 0000 N 174#
CC_PARITY_MSK 00E0 N 188#
CC_PARITY_SHF 0005 N 189#
CC_POL 0010 N 172#
CC_RTS 0004 N 170#
CC_SBITS_MSK 0C00 N 192#
CC_SBITS_SHF 000A N 193#
CC_SPACE 00C0 N 177#
CC_XON 0001 N 168#
CDB_AOLINK 0014 N 138# 666 750
CDB_COLINK 0012 N 137# 635 715
CDB_CURCHAR 0008 N 128#
CDB_CUROEM 000B N 130#
CDB_IINPUT 000F N 134# 518
CDB_INIT 0018 N 140# 503 577
CDB_INPUT 001A N 141# 689
CDB_INSTAT 001C N 142# 598
CDB_LOLINK 0016 N 139# 625 740
CDB_NAME 0000 N 126#
CDB_NFLAGS 0010 N 135#
CDB_OUTPUT 001E N 143# 727
CDB_OUTSTAT 0020 N 144# 648
CDB_RESRV 0011 N 136#
CDB_RXB 000D N 132#
CDB_SUPCHAR 0006 N 127#
CDB_SUPOEM 000A N 129#
CDB_TXB 000C N 131#
CDB_TYPE 000E N 133#
CHAIN_MSG 0154 V 462 924#
CMPERR_MSG 0184 V 465 933#
CON_LEN 0067 V 100#
CON_WIDTH 0066 V 99#
CS SREG V
CS_15_SBITS 4000 N 163#
CS_1_SBITS 2000 N 162#

```
CS_2_SBITS       8000 N   164#
CS_5_DBITS       0200 N   157#
CS_6_DBITS       0400 N   158#
CS_7_DBITS       0800 N   159#
CS_8_DBITS       1000 N   161#
CS_DTR           0008 N   151#
CS_ETX           0002 N   149#
CS_EVEN          0040 N   154#
CS_MARK          0080 N   155#
CS_ODD           0020 N   153#
CS_POL           0010 N   152#
CS_RTS           0004 N   150#
CS_SPACE         0100 N   156#
CS_XON           0001 N   148#
CT_INPUT         0001 N   197#
CT_OUTPUT        0002 N   198#
CT_SERIAL        0008 N   200#
CT_SOFTBAUD      0004 N   199#
DPB_AL0          0009 N   286#
DPB_AL1          000A N   287#
DPB_BLM          0003 N   282#
DPB_BSH          0002 N   281#
DPB_CKS          000B N   288#
DPB_DRM          0007 N   285#
DPB_DSM          0005 N   284#
DPB_EXM          0004 N   283#
DPB_OFF          000D N   289#
DPB_PHM          0010 N   291#
DPB_PSH          000F N   290#
DPB_SPT          0000 N   280#
DPH_ALV          000C N   253#
DPH_CHNNL        001D N   262#
DPH_CSV          000A N   252#
DPH_DATBCB       0010 N   255#
DPH_DIRBCB       000E N   254#
DPH_DOPEN        0005 N   250#
DPH_DPB          0008 N   251#
DPH_HSHTBL       0012 N   256#
DPH_INIT         0014 N   257#  510
DPH_LOGIN        0016 N   258#  772
DPH_NFLAGS       001E N   263#
DPH_READ         0018 N   259#  784
DPH_UNIT         001C N   261#
DPH_WRITE        001A N   260#  795
DPH_XLT          0000 N   249#
DS             SREG V
ERR1_MSG         01B1 V   468   942#
ERR2_MSG         01C7 V   469   946#
ERR3_MSG         01D6 V   470   949#
ERR4_MSG         01E5 V   471   952#
ERR5_MSG         01F3 V   472   956#
ERR6_MSG         0202 V   473   959#
ERR7_MSG         020E V   474   961#
ERR_MODE         0046 V   102#  882
```

```
ES              SREG V   102   555
FILE_MSG        01A8 V   467   940#
FUNCTAB         0130 V   554   905#
FUNC_MSG        0195 V   466   936#
INIT_DRV        0020 L   500   505#
INT_CHARSCAN    003C V    87#
INT_DISPATCH    0034 V    85#
INT_SETFLAG     0038 V    86#
IN_SCAN         00AF L   683#  701
IOPB_DMAOFF     0006 V   335#
IOPB_DMASEG     0008 V   334#
IOPB_DRIVE      000E V   331#  812
IOPB_MCNT       000F V   330#
IOPB_SECTOR     000A V   333#
IOPB_TRACK      000C V   332#
IO_AUXIN        00B9 L   695#  910
IO_AUXIST       0076 L   603#  921
IO_AUXOST       00A2 L   659#  922
IO_AUXOUT       00E1 L   744#  911
IO_CONIN        00AB L   675#  906
IO_CONIST       0068 L   586#  905
IO_CONOST       0085 L   628#  920
IO_CONOUT       00BF L   709#  888   907
IO_DEVINIT      0061 L   569#  919
IO_FLUSH        0128 L   821#  917
IO_LIST         00D8 L   734#  909
IO_LISTST       007C L   618#  908
IO_NOTIMP       012B L   837#  912   913   918
IO_READ         0108 L   777#  915
IO_SELDSK       00EA L   759#  914
IO_WRITE        010D L   787#  916
IST_SCAN        006C L   595#  610
NEXT_CHAR       014B L   884#  891
NEXT_DEVICE     0010 L   496#  515
NO_INPUT_DEV    00B6 L   688   691#
NO_STAT_DEV     0073 L   597   599#
OSBASESEG       0046 V    89#
OSENDSEG        0048 V    90#
OST_NOTRDY      00A1 L   650   656#
OST_RDY         009F L   646   654#
OST_SCAN        008C L   626   639#  653   667
OUT_EXIT        00D7 L   725   731#
OUT_SCAN        00C6 L   718#  730   741   751
PMSG_EXIT       0159 L   883   886   892#
PROMPT_MSG      016C V   463   928#
RANGE_ERR       005C L   550   559#
READ_WRITE      0112 L   785   797   805#
RET_ERROR       0125 L   815   817#
RLR             004E V    92#  555
SEL_RET         0105 L   767   770   771   773#
SS              SREG V
TOD_DAY         005F V    94#
TOD_HR          0061 V    95#
TOD_MIN         0062 V    96#
```

```
TOD_SEC          0063 V    97#
USER_STR          017F V   464   932#
ZERO_ENTRY         002D L   508   512#


EOF
```

CPMPIGC.WS4 (= "CP/M-86 Plus Installation Guide", appendix C)

(Edited by Emmanuel ROCHE.)

Appendix C: SYSDAT Format

This appendix discusses SYSDAT segment fields that are pertinent to the system implementor. The SYSDAT segment is the same as the BIOS data segment. The BIOS data starts at offset 0F00h, and the SYSDAT fields are at the beginning of the segment with the offsets shown in Figure C-1. Table C-1 describes each field.

```
      +-----+-----+-----+-----+-----+-----+-----+-----+
   28h:|      BIOS_ENTRY     |      BIOS_INIT      |
      +-----+-----+-----+-----+-----+-----+-----+-----+
   30h:|       BDOS          |    INT_DISPATCH     |
      +-----+-----+-----+-----+-----+-----+-----+-----+
   38h:|    INT_SETFLAG      |    INT_CHARSCAN     |
      +-----+-----+-----+-----+-----+-----+-----+-----+
   40h:|      RESERVED       | RESERVED  | OSBASESEG |
      +-----+-----+-----+-----+-----+-----+-----+-----+
   48h:| OSENDSEG |      RESERVED       |    RLR    |
      +-----+-----+-----+-----+-----+-----+-----+-----+
   50h:|               RESERVED                |
      +-----+-----+-----+-----+-----+-----+-----+-----+
   58h:|               RESERVED                |
      +-----+-----+-----+-----+-----+-----+-----+-----+
   60h:| TOD_DAY  | TOD | TOD | TOD | RESE|CON_ |CON_ |
      |          | _HR | _MIN| _SEC| RVED|WIDTH|LEN  |
      +-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure C-1. SYSDAT Fields

Listing C-1. SYSDAT.LIB

```
;*******************************************************
;
;       System Data Definitions
;
;*******************************************************

bdos            equ     dword ptr .30h  ;entry into operating system
int_dispatch    equ     dword ptr .34h  ;exit from interrupt handler
int_setflag     equ     dword ptr .38h  ;interrupt SETFLAG function
int_charscan    equ     dword ptr .3Ch  ;interrupt live key scanner

osbaseseg       equ     word ptr .46h   ;base of the OS in para's
osendseg        equ     word ptr .48h   ;first para after OS

rlr             equ     word ptr .4Eh   ;ready list root
```

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

con_width       equ     byte ptr .66h   ;console width
con_len         equ     byte ptr .67h   ;console length

err_mode        equ     es:byte ptr.46h ;process error mode in
                                ;in process descriptor
```

Table C-1. SYSDAT Fields

Format: Data Field
Explanation

BIOS_ENTRY
[Double word address of the JMP BIOSENTRY instruction in the BIOS Kernel Code Header] All BDOS calls to the BIOS go through this entry point.

BIOS_INIT
[Double word address of the JMP BIOSINIT instruction in the BIOS Kernel Code Header] Hardware initialization not performed by the disk boot loader or CPMLDR is performed by the BIOSINIT routine.

BDOS
[Double word address of the BDOS entry point for system calls made when already in the operating system] All BIOS calls back to the BDOS, except from interrupt service routines, use this double word address. See the public BIOS Kernel ?DISPATCH and ?DELAY routines for examples. The register conventions for this entry require DS to contain the SYSDAT segment (the BIOS data segment), and ES to contain the running process's environment segment. The other registers follow the conventions shown in the "Programmer's Guide" for a system call using INT 224 instruction, except that the BDOS puts the return code only in BX, and not in both AX and BX. The amount of stack required by the BDOS depends on the system call and the resulting BIOS functions that are required. However, performing an INT 224 instruction from the BIOS to invoke a system call requires more stack space. When the BIOS performs an INT 224, it is reentering the BDOS through the same path the application previously used. The BDOS only switches stacks on the first entry via an INT 224; otherwise, the same stack is used. Section 3 discusses the stack guaranteed by the BDOS on a first level call to the BIOS from the BDOS.

INT_DISPATCH
[Double word address of the BDOS dispatcher entry point]
INT_DISPATCH can only be used to exit an interrupt service routine. Executing a JMPF instruction to this address is equivalent to executing a POP DS and an IRET (Interrupt Return instruction). The dispatcher saves the context of the running process, and restores the context of a second process that has been waiting for the CPU. The dispatcher then gives the CPU to this second process. See "Interrupt Device Drivers" in Section 4.

INT_SETFLAG

[Double word address of the BDOS interrupt setflag function]
INT_SETFLAG can only be called from an interrupt service routine. See "Interrupt Device Drivers" in Section 4.

INT_CHARSCAN
[Double word address of the BDOS interrupt character scanner]
INT_CHARSCAN can only be called from an interrupt service routine. The character scanner is provided for live keyboard support of the physical device currently attached to the logical input device CONIN:. See "Interrupt Device Drivers" in Section 4, and "Interrupt-driven Character I/O" in Section 6.

OSBASESEG
[Starting paragraph of the operating system code]
OSBASESEG is set by GENCPM, and is the code segment of the BDOS. See Figure E-2.

OSENDSEG
[First paragraph past the end of the operating system data]
OSENDSEG includes all buffers allocated to the operating system by GENCPM, but not made part of the CPMP.SYS file. See "GENCPM Utility" in Section 9, and also Figure E-2.

RLR
[Ready list root]
RLR is the segment address of currently running process environment. The BDOS calls the BIOS with this value in the ES register, and the BIOS preserves or restores this value in ES before returning to the BDOS.

TOD_DAY
Number of days since January 1, 1978.

TOD_HR
Hour of the day in packed binary coded digits (BCD).

TOD_MIN
Minute of the hour in packed binary coded digits (BCD).

TOD_SEC
Second of the minute in packed binary coded digits (BCD).

CON_WIDTH
Number of columns of screen.

CON_LEN
Number of rows of screen.

EOF

CPMPIGD.WS4 (= "CP/M-86 Plus Installation Guide", appendix D)

(Edited by Emmanuel ROCHE.)

Appendix D: Disk Parameter Block Worksheet

This worksheet is intended to help in creating a Disk Parameter Block (DPB) containing the specifications for the particular disk hardware you are implementing. You can photocopy the DPB worksheet and use it to record your DPB calculations for each drive you define in the BIOS. Several of the steps in the worksheet represent intermediate calculations that are not part of the DPB. Steps that result in values to be placed in the DPB are labeled "field in Disk Parameter Block".

<A> Allocation Block Size

CP/M-86 Plus allocates disk space in a unit known as an allocation block. This is the minimum disk allocation unit for files on this drive. This value can be 1024, 2048, 4096, 8192, or 16,384 decimal bytes, or 400h, 800h, 1000h, 2000h, or 4000h bytes, respectively. Choosing a large allocation block size allows more efficient usage of directory space for large files and allows a larger number of directory entries. On the other hand, choosing a smaller block size increases the size of the allocation vectors since there are more blocks on an equivalent drive. A large allocation block size increases the average wasted space per disk file. This is the allocated disk space beyond the logical end of a file.

There are several restrictions on the block size. If the block size is 1024 bytes, there cannot be more than 255 blocks present on a logical drive. In other words, if the disk is larger than 256 Kbytes, it is necessary to use an allocation block size of at least 2048 bytes.

<B> DPB_BSH (Block Shift) field in Disk Parameter Block

<C> DPB_BLM (Block Mask) field in Disk Parameter Block

Determine the values of DPB_BSH and DPB_BLM from the following table given the allocation block size from step <A>.

Table D-1. DPB_BSH and DPB_BLM Values

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

<D> Total Allocation Blocks

Determine the total number of allocation blocks on the disk drive. First, calculate the total available space on the drive in bytes. Do this by multiplying the total number of tracks on the disk (minus reserved boot tracks) by the number of sectors per track and the physical sector size. Divide this figure by the allocation block size determined in <A> earlier. This quotient, rounded down to the next lowest integer value, is the total allocation blocks for the drive. (The boot tracks are determined by the DPB_OFF field.)

<E> DPB_DSM (Disk Size Max) field in Disk Parameter Block

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

<F> DPB_EXM (Extent Mask) field in Disk Parameter Block

Obtain the value of DPB_EXM from the following table, using the values from steps <A> and <E>.

Table D-2. DPB_EXM Values

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

<G> Directory Blocks

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

<H> Directory Entries per Block

Use the following table to determine the number of directory entries per directory block based upon the allocation block size from step <A>.

Table D-3. Directory Entries Per Block Size

<A>	Number of entries
1,024	32

2,048	64
4,096	128
8,192	256
16,384	512

<I> Total directory entries

Determine the total number of directory entries by multiplying the values found in steps <G> and <H>.

<J> DPB_DRM (Directory Max) field in Disk Parameter Block

Determine DPB_DRM by subtracting 1 from the value found in step <I>.

<K> DPB_AL0, DPB_AL1 (Directory Allocation vectors 0 and 1)

Determine DPB_AL0 and DPB_AL1 from the following table, given the number of directory blocks from in step <G>.

Table D-4. DPB_AL0, DPB_AL1 Values

<G>	DPB_AL0	DPB_AL1	<G>	DPB_AL0	DPB_AL1
1	80h	00h	9	0FFh	80h
2	0C0h	00h	10	0FFh	0C0h
3	0E0h	00h	11	0FFh	0E0h
4	0F0h	00h	12	0FFh	0F0h
5	0F8h	00h	13	0FFh	0F8h
6	0FCh	00h	14	0FFh	0FCh
7	0FEh	00h	15	0FFh	0FEh
8	0FFh	00h	16	0FFh	0FFh

<L> DPB_CKS (Checksum field) in Disk Parameter Block

Determine the size of the checksum vector. If the disk drive media is permanent, then set DPB_CKS to 8000h. If the disk drive media is removable, the value should be (<J>/4)+1. If the disk drive media is removable and if @BH_GDOPEN is set by a door open interrupt, DPB_CKS equals ((<J>/4)+1)+ 8000h. For removable media drives, the checksum vector is CKS-bytes long and is addressed in the DPH. When the Disk Parameter Header field DPH_CSV is set to 0FFFFh, GENCPM uses the DPB_CKS value to construct the checksum vector automatically.

<M> DPB_OFF (Offset) field in Disk Parameter Block

The DPB_OFF field determines the number of tracks that are skipped at the beginning of the physical disk. The BDOS automatically adds this to the value of IOPB_TRACK parameter and can be used as a mechanism for skipping tracks reserved for boot operations, or for partitioning a large disk into smaller

logical drives.

<N> Size of Allocation Vector

In the DPH, the allocation vector is addressed by the DPH_ALV field. The size of this vector is determined by the number of allocation blocks. The allocation vector is actually two separate concatenated bit vectors. The length of each vector is one bit per allocation block rounded up to the nearest byte. Thus, the size of allocation vector is equal to (<E>/4)+2. GENCPM uses the DPB_DSM value (step <E>) to automatically construct the checksum vector if the DPH_ALV field is set to 0FFFFh.

<O> Physical Sector Size

Specify the physical sector size of the disk drive. Note that the physical sector size must be 128, 256, 512, 1024, 2048, or 4096 bytes. The physical sector size must also be less than or equal to the allocation block size. If your sector size is not one of these values, you must perform blocking/deblocking to and from the sector size chosen in this step.

<P> DPB_PSH (Physical Record Shift) field in Disk Parameter Block

<Q> DPB_PHM (Physical Record Mask) in Disk Parameter Block

Determine the values of DPB_PSH and DPB_PHM from the following table, given the physical sector size from step <O>.

Table D-5. DPB_PSH and DPB_PHM Values

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

DPB Worksheet Parameter List

<A> Allocation Block Size _____

<B> DPB_BSH field in Disk Parameter Block _____

<C> DPB_BLM field in Disk Parameter Block _____

<D> Total Allocation Blocks _____

<E> DPB_DSM field in Disk Parameter Block _____

<F> DPB_EXM field in Disk Parameter Block _____

<G> Directory Blocks _____

<H> Directory Entries per Block _____

<I> Total directory entries _____

<J> DPB_DRM field in Disk Parameter Block _____

<K> DPB_AL0, DPB_AL1 fields in Disk Parameter Block _____

<L> DPB_CKS field in Disk Parameter Block _____

<M> DPB_OFF field in Disk Parameter Block _____

<N> Size of Allocation Vector _____

<O> Physical Sector Size _____

<P> DPB_PSH field in Disk Parameter Block _____

<Q> DPB_PHM field in Disk Parameter Block _____

EOF

CPMPIGE.WS4 (= "CP/M-86 Plus Installation Guide", appendix E)

(Edited by Emmanuel ROCHE.)

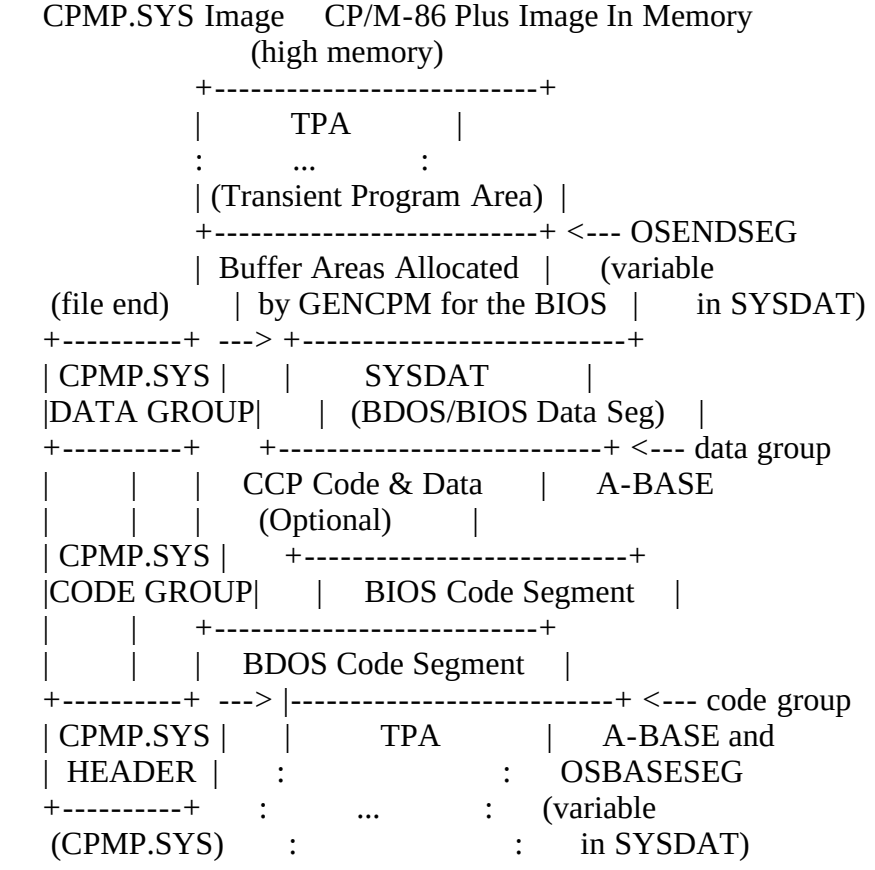
Appendix E: Memory Image and CPMP.SYS File

The CPMP.SYS file, generated by GENCPM and read by the CPMLDR, is prefixed by a 128-byte CMD file Header Record. (Appendix D in the "Programmer's Guide" presents more detail on the CMD file Header Record.) The CMD file Header Record contains the following two group descriptors:

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

Figure E-1. Group Descriptors in CPMP.SYS Header Record

The first group descriptor represents the code group of the CPMP.SYS file, and the second represents the data. The code and data A-BASE values are set by GENCPM, depending on the your answer to the "Base of CP/M-86 Plus" (Code) and the "Data Base of CP/M-86 Plus" (Data) questions (see Section 9). The entire CPMP.SYS file appears on disk as shown in Figure E-2.



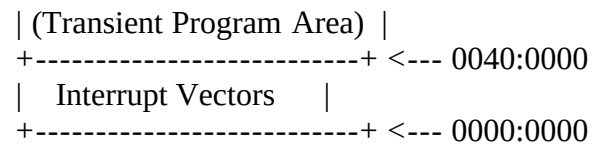


Figure E-2. CPMP.SYS File Image and CP/M-86 Plus Memory Image

CPMLDR reads into memory the CPMP.SYS file, beginning at the segment address given by code group A-BASE (OSBASESEG), which is found in the CMD Header, as shown in Figure E-1. CPMLDR sets the DS register to the value in the data group A-BASE field. Control is passed to the BDOS initialization code when CPMLDR executes a JMPF (Jump Far instruction) to OSBASESEG:0000h. Thus, the BDOS initialization routine starts with the CS register set to the code group A-BASE value, the IP register equal to zero, and the DS register equal to data group A-BASE value.

EOF

CPMPIGF.WS4 (= "CP/M-86 Plus Installation Guide", appendix F)

(Edited by Emmanuel ROCHE.)

Appendix F: Memory Descriptor Format

Memory Descriptors are three word structures kept in the BIOS Kernel Data Header. The Kernel Header reserves space for 32 Memory Descriptors, a total area of 192 bytes. GENCPM sets the first 8 Memory Descriptors to reflect the memory partitions you define. GENCPM initializes the rest of the Memory Descriptors for use by CP/M-86 Plus during memory allocation and de-allocation operations.

Figure F-1 shows the structure of each Memory Descriptor:

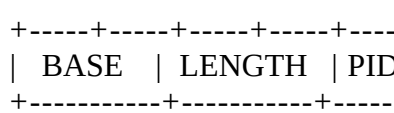


Figure F-1. Memory Descriptor Format

Table F-1 describes the Memory Descriptor format fields.

Table F-1. Memory Descriptor Format Fields

Format: Field	Explanation
BASE	Paragraph base (segment address) of this memory partition.
LENGTH	Length, in paragraphs, of this memory partition.
PID	If PID = 0FFh, the Memory Descriptor is unused. If PID = 0FEh, the Memory Descriptor describes a currently available memory partition. If PID is set to any other value, the memory described by the Memory Descriptor is allocated. GENCPM initializes the PID fields in all of the Memory Descriptors to 0FFh or 0FEh.
RESRV	Reserved for system use.

The free memory described by the Memory Descriptors is the size of the TPA at any given time. The memory manager submodule of the BDOS coalesces Memory Descriptors representing adjacent memory areas when memory is de-allocated. If

you do not want memory partitions to be coalesced, use GENCPM to define them as separated by one or more paragraphs. See Figure 9-6.

The BIOS INIT module can dynamically size memory at system boot time, and modify the Memory Descriptors. If the system's memory configuration can vary, the Memory Descriptors must be initialized by the BIOS INIT routine at boot time. If you need to locate the operating system image at BIOS initialization time, the start of the CP/M-86 code can be found in the SYSDAT variable OSBASESEG, and the paragraph after the CP/M-86 Plus data is found in the OSENDSEG variable. The beginning of CP/M-86 Plus data is the SYSDAT segment.

If there is no hardware support to determine the machine's memory configuration, memory can be sized by writing a pattern to the entire address space of the processor (excluding the operating system area), and reading it back to confirm the existence of RAM in each location. The example BIOS INIT module in the file BIOS.A86 on the distribution disks sizes memory in this way.

EOF

CPMPIGG.WS4 (= "CP/M-86 Plus Installation Guide", appendix G)

(Edited by Emmanuel ROCHE.)

Appendix G: Placing CP/M-86 Plus in ROM

The CP/M-86 Plus operating system was developed with separate code and data, to allow you to place CP/M-86 Plus in ROM. The contents of the CPMP.SYS file, code, and initialized data are placed in ROM, and at power-on, or hardware reset, the data is copied to RAM. This appendix assumes familiarity with the material covered in Sections 9 and 10 on using GENCPM and BIOS debugging.

You supply a "data mover" routine that receives control when the 8086 or 8088 is reset and copies the initialized data from the ROM to RAM. Figure G-1 shows one possible CP/M-86 Plus ROM image. In this example, at location 0FFFF:0000h is a JMPF (Jump Far instruction) to the START_MOVER: label at a lower memory location in the ROM. The data mover must exit by setting DS to the SYSDAT segment and performing a JMPF (Jump Far instruction) to the beginning of the BDOS code.

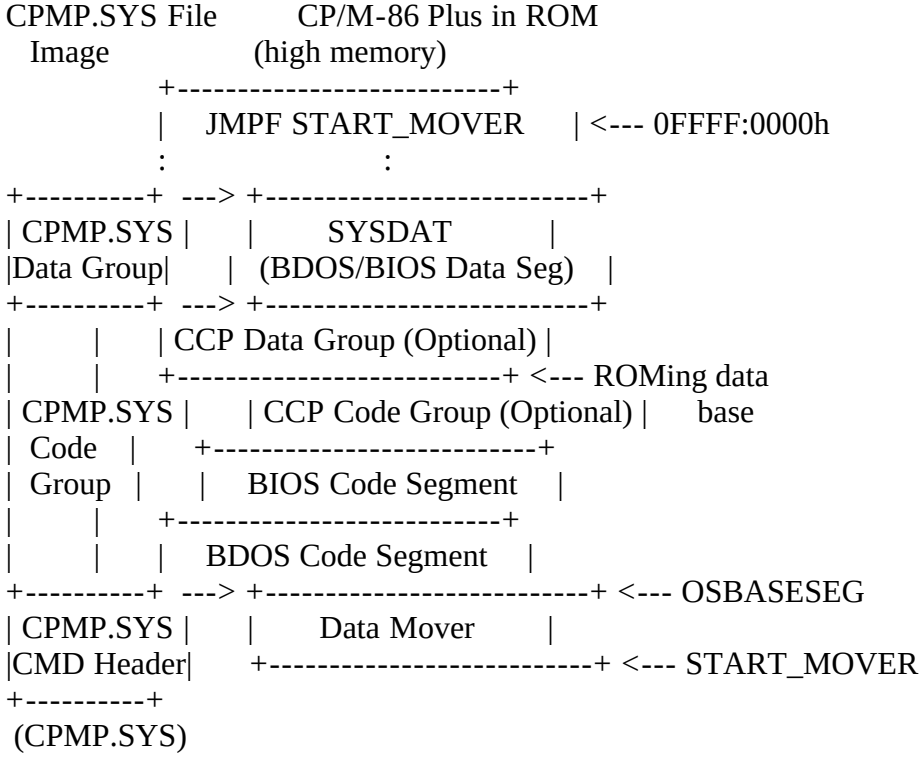
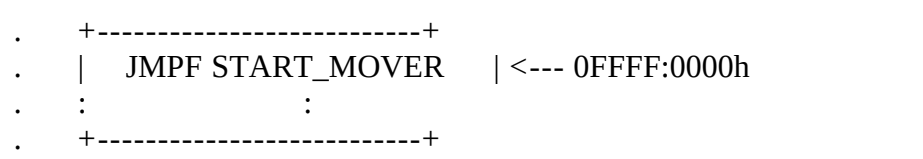


Figure G-1. An Example CP/M-86 Plus ROM Image

Figure G-2 shows the execution image after the initialized data has been moved to RAM.



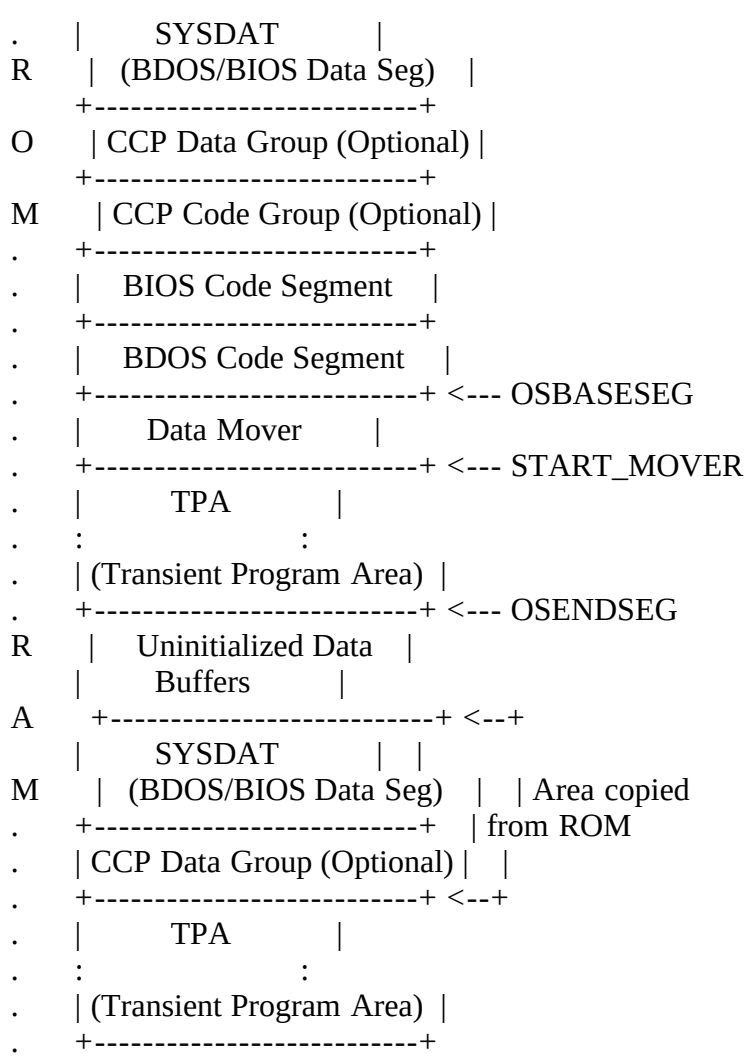


Figure G-2. CP/M-86 Plus Code in ROM and DATA in RAM

In Figure G-2, OSBASESEG is the starting paragraph of the operating system code, and OSENDSEG is the next paragraph after the operating system RAM area.

The size of the ROM required depends on the size of the BIOS, and also if you make the CCP a permanent part of the Operating System. The BDOS alone is about 21 Kbytes. The CCP is an additional 5 Kbytes.

The answer to the GENCPM "Code Base of CP/M-86 Plus" question (see Section 9) sets the segment address where CP/M-86 Plus system code must be located in RAM or ROM. The answer to the GENCPM "Data Base of CP/M-86 Plus" question (see Section 9) determines the RAM segment address where the uninitialized and initialized data areas must reside. The area in RAM reserved by GENCPM for the data must be large enough to contain both the initialized data and the uninitialized data required by the operating system. Note that the CPMP.SYS file created by GENCPM contains only the initialized data. See Figure E-2.

GENCPM displays the length in paragraphs of the operating system code, the initialized system data, and the total system data at the end of a GENCPM session (see Figure 9-8). The total system data is the sum of the initialized and uninitialized data. GENCPM resolves all references to the system data. GENCPM also adjusts the Memory Descriptors in the BIOS Kernel Data Header to exclude the data area reserved for the operating system. The following is an

example of the GENCPM display when the system code is set at segment 0F800h, and the system data at 40h:

```
      CP/M-86 Plus ROMing Information
                   Base   Length
                   ----   ------
    System Code            F800H  04F0H
    Initialized System Data 0040H  0179H
    Total System Data       0040H  0514H

    Operating System Memory Table:
    Partition  Base   Length
    -----------------------
      0      0554H  1AECH

    CPMP.SYS file created on drive D:

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

Listing G-1 shows an example data mover using this GENCPM information. At power-on or after a hardware reset, the hardware must transfer control to the START_MOVER: label.

Note: The SYSDAT value is found in the word at offset 6 within the BDOS code segment. When the CCP is made part of the system, the RAM data area begins with the CCP data, and not with the SYSDAT data segment.

Listing G-1. Example ROM Data Mover

```
; Construct a ROM image file using this program (DMOVER.A86),
; the CPMP.SYS file, and the following instructions:
;
;       A>rasm86 dmover         ;assemble this program
;       A>link86 dmover.sys=dmover
;       A>sid86
;       #rdmover.sys
;       START     END
;       ZZZZ:0000 ZZZZ:01FF      ;create a file containing
;       #wdmover,80,ff           ;the 1st 128 bytes of code
;       #^C                      ;from this program
;       A>sid86
;       #rcpmp.sys
;       START     END
;       ZZZZ:0000 ZZZZ:XXXX      ;strip the CMD file Header Record
;       #wcpmp,80,XXXX           ;from the CPMP.SYS file
;       #^C
;       A>pip rom.sys=dmover[o],cpm3[o]
;
;       The file ROM.SYS has the format:
;
; (file start)                    (file end)
; +------------+-------------+------------------------+
; | Mover Code | System Code | Initialized System Data |
; +------------+-------------+------------------------+
```

```
false   equ     0
true    equ     not false

sysdat  equ  word ptr .6

debug   equ  true

        if debug
SYSTEM_CODE         equ  2000h
        endif
        if not debug
SYSTEM_CODE         equ  0F800h
        endif

CODE_LENGTH         equ  4BAh          ;as displayed by GENCPM

INIT_DATA_LENGTH    equ  179h          ;as displayed by GENCPM

INIT_DATA_IN_ROM    equ  SYSTEM_CODE + CODE_LENGTH
INIT_DATA_IN_RAM    equ  3000h

        CSEG

start_mover:        ;JMPF instruction at 0FFFF:0000h goes here
;-----------
;       Entry:  none required
;               (after reset to 8086 or 8088
;               DS,ES,SS = 0, flags are reset, and
;               all other registers are unknown)
;       Exit:   DS = SYSDAT
;               JMPF's to start of BDOS

        mov ax,INIT_DATA_IN_ROM
        mov ds,ax                   ;start of data to move
        mov cx,INIT_DATA_LENGTH*8   ;words of initialized data
        mov ax,INIT_DATA_IN_RAM
        mov es,ax                   ;ES=RAM destination for
        xor si,si                  ;DS=ROM source of initialized data
        mov di,si                   ;DI=SI=0
        rep movsw                   ;copy from ROM to RAM
        mov ax,SYSTEM_CODE
        mov ds,ax
        mov ds,sysdat               ;get SYSDAT segment address
                                ;out of BDOS code
        jmpf cs: dword ptr bdos_init ;transfer control to BDOS

bdos_init       dw      0
                dw      SYSTEM_CODE
```

The example data mover is made into a 128-byte file by the SID-86 command shown in the comment that begins the mover listing. The 128-byte length is used for simplicity and ease of manipulation. Your data mover can certainly be smaller if need be. After the initialized data is moved, the data mover must

set the DS register to the SYSDAT segment, then perform a JMPF to the beginning of the BDOS code. The SYSDAT data segment address is found at offset 6 within the BDOS code segment.

To debug a data mover similar to the example in Listing G-1 under CP/M-86 1.X or CP/M-86 Plus, create the ROM.SYS file where SID-86 or DDT-86 can read and execute it in RAM. Set the "Code Base of CP/M-86 Plus" to this location in RAM, allowing 8 paragraphs for the data mover. Set the "Data Base of CP/M-86 Plus" to a RAM area large enough for the uninitialized and initialized data. Additionally, the data mover must be assembled reflecting the segment address of where the data is to be copied. Listing G-1 includes a debug toggle for testing in RAM. The layout of memory for debugging the ROM.SYS file is similar to the layout for debugging the CPMP.SYS file, except the CMD file Header Record of the CPMP.SYS file is replaced with the data mover. Section 10 discusses the debugging of the CPMP.SYS file.

When your data mover works in RAM, run GENCPM, specifying the segment in ROM where the system code is to reside. Answer the "Data Base of CP/M-86 Plus" question with desired segment address in RAM for the initialized and uninitialized system data. If you have a debug toggle in your data mover routine, set it to false and reassemble it. Place the CPMP.SYS file and the data mover in ROM, such that the data mover receives control on power-on or hardware reset.

EOF

CPMPIGH.WS4 (= "CP/M-86 Plus Installation Guide", appendix H)

(Edited by Emmanuel ROCHE.)

Appendix H: Foreign Language Messages

All English messages CP/M-86 Plus displays can be modified or translated to foreign languages. This appendix describes the procedures for changing these messages or replacing them with the translations you supply. Error messages and utility options, as well as the headers for tabular displays, can all be altered.

The text strings CP/M-86 Plus displays come from two sources. The first is from the BDOS and the BIOS modules contained in the system image. The second source is from the Digital Research utilities. The translation of any BIOS messages is the responsibility of the system implementor.

CUSTOMIZING BDOS MESSAGES

The text strings the BDOS displays are all defined in the BIOS Kernel. To change messages, you edit the Kernel, reassemble it, and generate a new system as outlined in Section 9. The BDOS prints file-related error messages, a chain error message, and the Error CCP prompt. The BDOS does not print file-related error messages if the program that encountered the error is running in Return Error Mode. See Section 3 and F_ERRMODE system call in the "Programmer's Guide". The BDOS displays the chain error message when a program calls the P_CHAIN system call and the BDOS has released the calling program's memory when the error is encountered. (The CCP displays a similar error when control can be returned to the calling process on a P_CHAIN call.) The Error CCP, described in the "User's Guide", displays a prompt message, and uses a string to recognize its one internal function.

The offsets of the strings used by the BDOS are contained in the BIOS Kernel Data Header. Listing H-1 shows this part of the Data Header. "BIOS Kernel Data Header" in Section 3 describes each field. These messages are defined at the end of the BIOS Kernel shown in Appendix B.

Listing H-1. BIOS Kernel Data Header Text Offsets

```
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 command string
bh_cpmerr       dw      cmperr_msg    ;CP/M error message address
bh_func         dw      func_msg      ;function message address
bh_file         dw      file_msg      ;file message address
bh_err1         dw      err1_msg      ;file related errors
bh_err2         dw      err2_msg      ;1-7
```

```
bh_err3     dw    err3_msg
bh_err4     dw    err4_msg
bh_err5     dw    err5_msg
bh_err6     dw    err6_msg
bh_err7     dw    err7_msg
```

The carriage returns (decimal 13) and the linefeeds (decimal 10) that are part of the string definitions in the Kernel should be preserved in any modifications you make. The termination character $ must also be preserved.

File-related error messages are printed in the following form:

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

This list shows how the BDOS prints these error messages:

1. The string associated with BH_CPMERR field is displayed. The default definition is 13,10,'CP/M Error On $'.

2. The drive spec "d:" is printed, which is one of the logical drives A:-P:.

3. Next, the "file_error_message" is printed. This is one of the seven messages addressed by BH_ERR1, BH_ERR2, ... BH_ERR7.

4. The BDOS prints the string whose offset is contained in the BH_FUNC field. The default definition is 13,10,'BDOS Function = $'.

5. The system call number "xx" is printed. This is the last system call the program made by performing an INT 224 (Interrupt instruction).

6. The string addressed by BH_FILE is printed. The default definition is ' File = $'.

7. Finally, the file name and file type that make up the filespec are printed by the BDOS.

CUSTOMIZING UTILITY MESSAGES

The following subsection describes the process of customizing the text strings that are displayed by CP/M-86 Plus utilities (CMD files). The distribution disks contain the object module files for all CP/M-86 Plus utilities, plus two libraries containing the messages the utilities display. To create a CMD file for a specific utility, the utility's OBJ file or files are linked with the libraries of messages. The customization of utility messages consists of the generation of new libraries of messages, then "relinking" the utilities.

Each utility has a set of external symbols that must be resolved at link time to reference the correct message. Some utilities share messages. This occurs whenever the same externally defined symbol appears in more than one utility. LINK-86 allows public symbols to be defined only once; thus, the message

libraries can contain just one definition for any one external message symbol.

There are approximately 500 different message symbols and their strings that must be defined by the libraries. Each public message symbol and its message must be assembled in a separate file. The OBJ files for each message must then be placed in one of the message libraries. A procedure for modifying the utility messages using several special programs and submit files is discussed later in this section. The messages must be in separate modules in the library files, to allow LINK-86 to include only the messages required in the utility CMD file. Two separate libraries are necessary, since LIB-86 allows a maximum of 256 modules.

The following steps form the procedure for altering the utility messages:

1. Edit the message files to contain the new text strings.

2. Create the libraries by running the STRIP.CMD program and the RASMLIB.SUB submit sequence.

3. Link the utilities to the new messages using the UTILITY.SUB submit.

The MESSG1.TXT and MESSG2.TXT files contain all the symbols used to generate the two libraries, MESSG1.L86 and MESSG2.L86. Use a standard text editor to modify or translate the strings in these two ASCII TXT files. The format of each symbol name and its string definition must be in the following form. Note that the text string must be defined by single quotations.

```
message_name    DB      'text string'<CRLF>
;;<CRLF>
```

The message_name cannot be modified, since this is the external symbol name. Since each line becomes part of a file assembled by RASM-86, the 'text string' follows the rules for the RASM-86 DB directive.

Each symbol and its string definition must be separated from the next symbol and its string definition by a carriage return, line feed, two semicolons, and another carriage return and line feed. The STRIP utility uses the <CRLF>;;<CRLF> sequence to recognize each symbol definition. (The last message of the file need not end in the <CRLF>;;<CRLF> sequence.) The following symbol definitions are from the MESSG1.TXT file.

```
msg0090 db      'Directory full - $'
;;
msg0175 db      'File not found: $'
;;
```

A valid modification to these messages could be the following German translation:

```
msg0090 db      'Speicher voll - $'
;;
msg0175 db      'Dokument besteht nicht: $'
;;
```

The $ sign is the delimiter for these two messages, and must be preserved. Many messages such as these two examples precede file or drive specifications that are printed by the utility, thus the " - " and the ": " strings are also preserved in the example translation.

Generally, knowing the context in which a specific error message is produced helps you to decide the appropriate size of the new message. If a message ends with a specific delimiter, usually a 0 or a $, you must preserve it.

The first message file, MESSG1.TXT, contains the error messages in the order in which Appendix D of the "User's Guide" discusses them. Since not only the utility error messages can be modified, but also the informational messages and the headers for tabular displays produced by the utilities, the CMDMESSG.TXT file is included on the distribution disks. The CMDMESSG.TXT file lists all the utilities and the strings produced by them. Print this file and use it as a reference along with Appendix D in the "User's Guide" when modifying the utility strings. It is also helpful to print for reference the two symbol definition files, MESSG1.TXT and MESSG2.TXT, as they are unmodified on the distribution disks.

The MESSG1.TXT and MESSG2.TXT files, and the corresponding libraries MESSG1.L86 and MESSG2.L86, are organized as follows:

- MESSG1.L86
  This file contains all the error messages (MSG0000 - MSG0380) for all the utilities. It also contains the strings for BACK, CCP, DATE, DIR, ERASE, GET, INITDIR, PIP, PUT, SHOW, SUBMIT, and TYPE, plus miscellaneous messages and strings.

- MESSG2.L86
  This file contains the strings for DEVICE, GENRSX, HELP, SET, and SETDEF, as well as additional strings for the DIR utility.

The STRIP program starts a cycle of creating an A86 file, then invoking a submit job to assemble and place the new message definition in one of the two library files. STRIP copies one symbol definition from the MESG.TXT, and inserts it, along with the public declaration syntax and any other information required by RASM-86, into a source file named using the first 8 characters of the symbol name. STRIP uses the file named MESG.TXT for the symbol and string definitions. You can copy the MESSG1.TXT or MESSG2.TXT files to MESG.TXT, or create a MESG.TXT file with only the symbol definitions you are currently working on. STRIP saves its location in the MESG.TXT file in the temporary file SAVE.$$$. STRIP then makes a P_CHAIN system call to SUBMIT, specifying the RASMLIB.SUB submit file and a single parameter, which is the first 8 characters of the symbol name. RASMLIB.SUB contains commands to assemble and place the new message OBJ module in one of the message library files. The first 8 characters in the symbol name inform LIB-86 which module to replace in the library. A command in the submit file erases the files named using the first 8 characters of the symbol which were created by STRIP and RASM-86. The last line in the RASMLIB.SUB file reinvokes the STRIP program. STRIP reads the SAVE.$$$ file to find the next symbol definition to copy from the MESG.TXT file. The SAVE.$$$ file consists of one word value at offset 0 in the file.

This value is the number of the next symbol definition for STRIP to process.

If you create a MESG.TXT file containing only the two example symbol definitions shown earlier, STRIP would first create the file MSG0090.A86 containing the following:

```
DSEG
PUBLIC msg0090

msg0090 db      'Speicher voll - $'
;;
```

STRIP constructs the name MSG0090.A86 from the first 8 characters of the symbol name. STRIP then saves the number of messages processed in the file SAVE.$$$, which at this point would be 1. Next, it chains to SUBMIT and specifies RASMLIB as the submit file, and the first 8 letters of the symbol name as the only parameter to the submit. The chain command buffer is the following:

```
SUBMIT RASMLIB MSG0090
```

The RASMLIB.SUB file must contain the following commands when you are modifying the MESSG1.L86 library:

```
RASM86 $1
LIB86 MESSG1 = MESSG1.L86 [REPLACE[$1]]
ERASE $1.*
STRIP
```

The example chain command used by STRIP results in the following commands being executed by SUBMIT:

```
RASM86 MSG0090
LIB86 MESSG1 = MESSG1.L86 [REPLACE[MSG0090]]
ERASE MSG0090.*
STRIP
```

When you are modifying messages in the MESSG2.L86 library, the RASMLIB.SUB file should contain the following commands:

```
RASM86 $1
LIB86 MESSG2 = MESSG2.L86 [REPLACE[$1]]
ERASE $1.*
STRIP
```

You can edit the RASMLIB.SUB file, or create it using PIP and one of two files on the distribution disks, RASMLIB1.SUB or RASMLIB2.SUB. The STRIP program is in source form on the distribution disks in the file STRIP.A86, if you must modify it. For a complete description of RASM-86 and LIB-86, please consult the "Programmer's Utilities Guide". Usually, you should run STRIP and record what it does by using the PUT command to echo console output to a file, or by using the Ctrl-P function to echo console output to the printer. If STRIP, RASM-86, or LIB-86 encounter an error while processing the MESG.TXT file, you can stop STRIP with a Ctrl-C, correct the problem, and restart STRIP. To do

this, read the SAVE.$$$ using the R (Read) command under one of the debuggers (SID-86 or DDT-86), decrement the first word in the file with the SW0 (Set Word) command, then use the W (Write) command to update the SAVE.$$$ file on disk. When you now invoke STRIP, it continues from the prior symbol definition in the MESG.TXT file. You can also set the SAVE.$$$ file to start STRIP with any symbol definition in the MESG.TXT file. The lowest value in the SAVE.$$$ word can be 0, corresponding to the first symbol definition.

Alternatively, you can redefine a symbol and its message by manually performing the assembly and library commands. This procedure is useful when you have a few errors and do not want to process a large symbol definition file again. For example, if MSG0090 did not have a closing quote mark, fix the main file, and create MSG0090.A86. Proceed to execute the individual steps of the RASMLIB.SUB file for the MSG0090 files.

It is a good idea to keep master symbol definition files for each of the two libraries. If you need to generate an entire library, you can copy the master file for the library to the MESG.TXT file. Then, copy to the RASMLIB.SUB file either the RASMLIB1.SUB or RASMLIB2.SUB files, depending on the library being created. Finally, invoke STRIP. Whenever you intend to define all of the symbols contained in the current MESG.TXT file, ensure the SAVE.$$$ file is erased before starting STRIP.

Some trial and error is necessary when testing new messages. The closer the new messages are to the original format, then the easier the translation or modification process. Testing can be accomplished quickly by making incremental changes to the libraries. Once you have working symbol definitions, be sure to update your master symbol definition file for the appropriate library.

In summary, perform the following steps to update a message symbol library after editing the symbol definition file:

1. Erase SAVE.$$$.

2. Ensure that MESG.TXT contains the correct messages.

3. Run PUT to maintain a log of the output of STRIP, or activate printer echo with Ctrl-P.

4. Run STRIP.

5. Verify that each message was correctly processed.

6. Update your master symbol definition file for the appropriate library.

The following items outline some conventions and restrictions in modifying the utility messages.

- If the message is a column heading, the column width governs the maximum length of substrings in the heading. For example, the output of SHOW [LABEL] uses the strings associated with the symbols SHO_LINE1 through SHO_LINE4. (See the symbol definition file MESSG1.TXT.) The

command SHOW [LABEL] on a drive with a directory label results in the display of the following header:

```
Directory     Passwds  Stamp   Stamp
Label         Reqd     Create  Update  Label Created   Label Updated
------------  -------  ------  ------  --------------  --------------
```

The substring 'Stamp', which is part of the SHO_LINE1 definition, labels a column six characters wide. Thus, if 'Stamp' is changed, the new string must not exceed six characters in length.

- The command line options to all the utilities are included in the libraries. One string contains all the options separated by delimiters, and another string of bytes is an array of offsets into the option string. The options can be renamed and their length changed as desired, but the start of each option in the string must be updated in the respective offset array. For example, the ERASE utility uses the options list symbol ERA_OPT and its definition 'XFCB0CONFIRM',0FFh. ERASE uses the symbol ERA_OFF and its array of offsets 0,5,12 to locate the start of XFCB, CONFIRM, and 0FFh. If the options are changed to 'PASSWORD0CONFIRM',0FFh, then the ERA_OFF symbol must be defined as 0,9,16. The semantics of the options are dependent on their position in the option string. Note that 0FFh marks the end of the option lists for different utilities, that _OPT is appended to option list symbols, and _OFF appended to symbols associated with the option list offsets. The symbols ERA_OPT and ERA_OFF are defined in the MESSG1.TXT file.

- Most options are separated by the number 0, as just shown for ERASE. PUT, GET, and DEVICE are exceptions using the tilde (~) to separate their options. These utilities use the symbol PUT_EOSMARK, GET_EOSMARK, and DEV_EOSMARK to define the tilde option separator. It is easiest to maintain the same separators, but if you must change them, you must also change the corresponding symbol suffixed with _EOSMARK for the corresponding utility.

- The delimiters each utility uses are defined by the symbols suffixed with _DLM, for instance, ERA_DLM and PUT_DLM. These characters are likely to change with different keyboards and international character codes. As with the option lists, the position in the delimiter string defines the semantics of the delimiters. If left and right brackets, [ ], are replaced with some other option delimiters, then the original order in the delimiter string must be preserved. For instance, in the string associated with ERA_DLM, the default delimiters are 0,'[]=, ',0,0FFh. These can become 0,'{}=, ',0,0FFh, but the positions of left and right brackets are maintained.

  The user interface should be consistent. Thus, if you change the option delimiters [ ] in one utility, they should be replaced with the same option delimiters for the rest of the utilities. Note that the CCP option symbol is named CCP_OPTSYM.

- Four symbols and their default strings that are of major importance for obvious reasons are MSG_LYES, defined as 'yes'; MSG_UYES, defined

as 'YES'; MSG_LNO, defined as 'no'; and MSG_UNO defined as 'NO'.

- The ASCII character set is the basis for all string processing in CP/M-86 Plus. Translation from lowercase to uppercase by subtracting 20h, is made if the character is in the range of a-z. This restriction is important when translating to alphabets with more than 26 letters.

  Furthermore, CP/M internals and standards are still expected. For example, drives are still labeled from A-P, the colon (":") is used as a drive separator, the semicolon (";") in the file specification serves as a password delimiter, and so forth. See the "Programmer's Guide" and the "User's Guide" for more information on file specifications.

- Do not use the RASM-86 $NC (No Case) option. This option causes symbols to be created in the OBJ file in whatever case they appear in the source file. Without this option, the symbols are translated to uppercase in the OBJ file. The external symbol declarations in the utility OBJ files are in uppercase.

- The messages produced by the ED.CMD utility cannot be customized.

- The BACK utility options, defined by the symbol BAK_OPTS, must be unique to BACK, and cannot be shared by any other utility. BACK removes any options in the command tail that match those in the BAK_OPTS definition. Then BACK passes the command tail to the program to which it chains.

- The messages for GENCPM are found in the GENMSG.TXT and the corresponding library file GENMSG.L86. To change the GENCPM messages, you modify the GENMSG.TXT file, copy it to MESG.TXT, copy the RASMLIBG.SUB file to RASMLIB.SUB, and then invoke STRIP.

- The HELP.HLP file can modified as show in the "User's Guide" to change the text displayed by HELP.CMD. The messages displayed by the HELP utility itself, however, are modified as outlined in this section.

The UTILITY.SUB file on the distribution disks contains the commands to link the utility object modules and the message libraries to create the utility CMD files. For example, the SET.CMD file is created with the command from the UTILITY.SUB file:

```
LINK86 SET = SCD2,SET,MESSG1.L86[S],
MESSG2.L86[S,DATA[ORIGIN[0],MAX[0]]]
```

The filetype (L86) signals LINK-86 that the MESSG files are libraries. The S (SEARCH) option specifies to LINK-86 to only include the modules necessary to resolve external references, and not the entire library. The file SCD2.OBJ is an interface module between the utilities and CP/M-86 Plus.

EOF

CPMPIGI.WS4 (= "CP/M-86 Plus Installation Guide", appendix I)

(Edited by Emmanuel ROCHE.)

Appendix I: Files on Distribution Disks

This appendix describes the several sets of files contained in the OEM version of CP/M-86 Plus:

- BDOS3.SYS and LBDOS.SYS are binary image files, containing the operating system kernel and the loader kernel.

- The source files used to construct the example BIOS for the CompuPro 8/16 are the following:

  BIOSKRNL.A86, CHARIO.A86, DISKIO.A86, CLOCK.A86, INIT.A86, SYSDAT.LIB, DISK.LIB, CDB.LIB, PIC.LIB

- RW64.A86 is an example source file not used in the CompuPro BIOS. RW64.A86 illustrates Multisector I/O for machines that cannot perform disk reads or writes over 64 Kbyte segment boundaries.

- The following files are also used in BIOS construction (see Section 9):

  MODEDIT.CMD, GENCPM.CMD, ZERO.L86

- The following source files are used to construct the example disk boot header and loader for the CompuPro 8/16 (see Section 11):

  DSKBOOT.A86, LPROG.A86, LBIOS.A86, TCOPY.A86

- GENLDR.CMD is used to construct the CPMLDR.SYS file.

- The following programmer's utilities are provided for system implementation:

  RASM86.CMD, LINK86.CMD, LIB86.CMD, XREF86.CMD, SID86.CMD, DDT86.CMD

- The following files are used to modify the utility messages (see Appendix H):

  MESSG1.TXT, MESSG1.L86, MESSG2.TXT, MESSG2.L86, GENMSG.TXT, GENMSG.L86, RASMLIB1.SUB, RASMLIB2.SUB, RASMLIBG.SUB, STRIP.A86, STRIP.CMD, CMDMESSG.TXT, UTILITY.SUB

- Several utilities are distributed to the OEM in source form. COPYDISK.A86 is written for the CompuPro 8/16, and is intended as an example for similar utilities on your hardware. ANYRSX.A86 is an RSX that is made part of the example COPYDISK utility (see Section 12). DEVICE.PLM is included in source form, since it manipulates the BIOS

Character Device Blocks in the BIOS. If you define special characteristics in the CDB, you can modify DEVICE to display and set them. DEVICE is written in PLM, and the OEM must provide a PLM development environment, or translate DEVICE to another language. DUMP-86 is an example assembly level language program that can be distributed to end-users. The CCP.A86 (Console Command Processor) is the user interface, or shell. You can rewrite the CCP and replace it, if needed. The distributed CCP expects certain files to be on disk and accessible through the drive search chain. These files are DIR.CMD, ERASE.CMD, RENAME.CMD, TYPE.CMD, GETRSX.RSX, and PUTRSX.RSX.

- The following utilities are distributed to the OEM in CMD form. The OBJ and other files used to create these utilities are also listed. Use these files to re-link the utilities after the utility message libraries are altered.

BACK.CMD, BACK.OBJ, BACK49.OBJ, GETF.OBJ, GETRSX.RSX, PUTF.OBJ, PUTRSX.RSX
CCP.CMD, CCP.OBJ
DATE.CMD, DATE.OBJ
DEVICE.CMD, DEVICE.OBJ
DIR.CMD, MAIN.OBJ, DISP.OBJ, SCAN.OBJ, DPB86.OBJ, SEARCH.OBJ, SORT.OBJ, TIMEST.OBJ, UTIL.OBJ
DDT86.CMD (messages cannot be customized)
DUMP86.CMD
ED.CMD (messages cannot be customized)
ERASE.CMD, ERASE.OBJ
GENCPM.CMD, GENCPM.OBJ, SETBOF.OBJ, GETDEF.OBJ, CRDEF.OBJ, GENDATA.OBJ
GENRSX.CMD, GENRSX.OBJ
GET.CMD, GET.OBJ, GETF.OBJ, GETRSX.RSX
HELP.CMD, HELP.OBJ
INITDIR.CMD, INITDIR.OBJ, ANYRSX.OBJ, INITDIRA.OBJ, DIOMOD.OBJ, PRTMSG.OBJ
PATCH86.CMD, PATCH86.OBJ
PIP.CMD, PIP.OBJ, INPOUT.OBJ
PUT.CMD, PUT.OBJ, PUTF.OBJ, PUTRSX.RSX
RENAME.CMD, RENAME.OBJ
SET.CMD, SET.OBJ
SETDEF.CMD, SETDEF.OBJ
SHOW.CMD, SHOW.OBJ, SHOWF.OBJ
STOP.CMD, STOP.OBJ, STOPF.OBJ
SUBMIT.CMD, SUBMIT.OBJ, GETF.OBJ, SUBRSX.RSX
TYPE.CMD, TYPE.OBJ

- The SCD2.OBJ file is an interface module between the utilities and CP/M-86 Plus. The HELP.HLP file contains the text for the HELP utility. You can modify the file as documented in the "User's Guide".

EOF