DIGITAL RESEARCH®

CP/M-8000™

Operating System

System Guide

1038-2013-001

CP/M-8000™

Operating System

System Guide

COPYRIGHT

DISCLAIMER

NOTICE TO USER

From time to time changes are made in the filenames and in the files actually included on the distribution disk. This manual should not be construed as a representation or warranty that such files or facilities exist on the distribution disk or as part of the materials and programs distributed. Most distribution disks include a "README.DOC" file. This file explains variations from the manual which do constitute modification of the manual and the items included therewith. Be sure to read this file before using the software.

TRADEMARKS

CP/M, CP/M-86, and Digital Research and its logo are registered trademarks of Digital Research. CP/M-68K, CP/M-80, CP/M-8000, DDT, MP/M, and TEX are trademarks of Digital Research. Zilog is a registered trademark of Zilog, Inc. Z80 and Z8000 are trademarks of Zilog, Inc. Olivetti is a registered trademark of Olivetti, Inc. PCOS and Olivetti M20 are trademarks of Olivetti, Inc. TI Silent 700 Terminal is a registered trademark of Texas Instruments, Incorporated.

The CP/M-8000™ Operating System System Guide was prepared using the Digital Research TEX™ Text Formatter and printed in the United States of America.

****
* First Edition: August 1984 *
****

Foreword

CP/M-8000™ is a single-user general purpose operating system. It is designed for use with any disk-based computer using a Zilog® Z8000™ or compatible processor. CP/M-8000 is modular in design, and can be modified to suit the needs of a particular installation.

The hardware interface for a particular hardware environment is supported by the OEM or CP/M-8000 distributor. Digital Research® supports the user interface to CP/M-8000 as documented in the CP/M-8000 Operating System User's Guide. Digital Research does not support any additions or modifications made to CP/M-8000 by the OEM or distributor.

Purpose and Audience

This manual is intended to provide the information needed by a systems programmer in adapting CP/M-8000 to a particular hardware environment. A substantial degree of programming expertise is assumed on the part of the reader, and it is not expected that typical users of CP/M-8000 will need or want to read this manual.

Prerequisites and Related Publications

In addition to this manual, the reader should be familiar with the architecture of the Zilog Z8000 as described in the Zilog 16-Bit Microprocessor User's Manual (third edition), the CP/M-8000 Operating System User's Guide, the CP/M-8000 Operating System Programmer's Guide, and, of course, the details of the hardware environment where CP/M-8000 is to be implemented. Further information on assembly language programming for the Z8000 may be found in Programming the Z8000, by Richard Mateosial, Sybex, 1980.

How This Book is Organized

Section 1 presents an overview of CP/M-8000 and describes its major components. Section 2 discusses the adaptation of CP/M-8000 for your specific hardware system. Section 3 discusses bootstrap procedures and related information. Section 4 describes each BIOS function including entry parameters and return values. Section 5 describes the process of creating a BIOS for a custom hardware interface. Section 6 discusses how to get CP/M® working for the first time on a new hardware environment. Section 7 provides information on using the distributed version of CP/M-8000. Section 8 describes the PUTBOOT utility, which generates a bootable disk.

Appendix A describes the contents of the CP/M-8000 distribution disks. Appendix B is a listing of the normal and boot BIOS's, conditionally compiled. Appendix C contains a listing of the PUTBOOT utility program.

Table of Contents

1 System Overview

2 System Generation

3 Bootstrap Procedures

Table of Contents
(continued)

4 BIOS Functions

5 Creating a BIOS

6 Installing and Adapting the Distributed BIOS and CP/M-8000

7 Cold Boot Automatic Command Execution

8 The PUTBOOT Utility

Appendixes

Tables and Figures

Tables

Figures

Section 1
System Overview

1.1 Introduction

CP/M-8000 is a single-user, general purpose operating system for microcomputers based on the Zilog Z8000 or equivalent microprocessor chip. It is designed to be adaptable to almost any hardware environment, and can be readily customized for particular hardware systems.

CP/M-8000 is equivalent to other CP/M systems with changes dictated by the Z8000 architecture. In particular, CP/M-8000 supports the very large segmented address space of the Z8000 family.

The CP/M-8000 file system is upwardly compatible with CP/M-80™ Version 2.2, CP/M-86™ Version 1.1, and CP/M-68K™ Version 1.2. The CP/M-8000 file structure allows files of up to 32 megabytes per file. CP/M-8000 supports from one to sixteen disk drives with as many as 512 megabytes per drive.

The entire CP/M-8000 operating system resides in its own memory segment at all times, and is not reloaded at a warm start. CP/M-8000 can be configured to reside in any portion of memory. The remainder of the address space is available for applications programs, and is called the transient program area, TPA. The TPA is assumed to consist of one or more complete (64 Kbyte) memory segments. CP/M-8000 supports both segmented and non-segmented user programs, and supports the the splitting of user program and data into separate addressing spaces.

Several terms used throughout this manual are defined in Table 1-1.

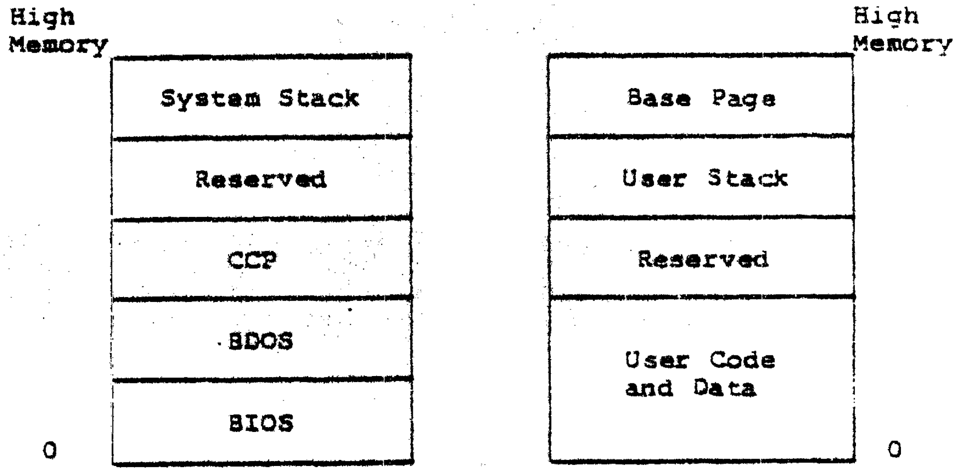
Table 1-1. CP/M-8000 Terms

Term	Meaning
nibble	4-bit half-byte
byte	8-bit value
word	16-bit value
longword	32-bit value
address	32-bit identifier of a storage location
physical address	address of a location in physical memory

Table 1-1. (continued)

Term	Meaning
logical address	address as issued by a program, possibly requiring translation into a physical address.
system mode	a program running in system mode can execute all instructions, including I/O instructions and instructions to change the contents of special control registers
normal mode	programs running in normal mode are prevented from executing the so-called privileged instructions
offset	a value defining an address in storage; a fixed displacement from a base address. For example, the base address of segment AH with an offset of 8000H provides a physical address of 0A008000H.
text segment	program section containing machine instructions
data segment	program section containing initialized data
block storage	program section containing uninitialized data
segment (Z8001)	set of adjacent memory addresses (up to 64K) with the same segment number
segmented mode	running-state of the segmented CPU in which addresses can have different segment members
non-segmented mode	running-state of the Z8000 CPU's. All addresses generated by segmented CPU's in this mode have the same segment number
absolute	describes a program that must reside at a fixed memory address.
relocatable	describes a program which includes relocation information so it can be loaded into memory at any address

The CP/M-8000 programming model is described in detail in the *CP/M-8000 Operating System Programmer's Guide*. After CP/M-8000 is loaded in memory, the remaining segments of address space that are not occupied by the operating system are called the Transient Program Area (TPA). To summarize this programming model briefly, CP/M-8000 supports the following memory segments that are not occupied by the operating system: a user stack, a base page and the three program segments. These three program segments consist of a text segment, an initialized data segment , and a block storage segment (bss). When a program is loaded, CP/M-8000 allocates space for these program segments in the TPA. The BDOS Program Load Function (59) loads a transient program in the TPA. If memory locations are not specified when the transient program is linked, the program is loaded in the TPA as shown in Figure 1-1.



CP/M System Memory (System Address Space)

Transient Program Area (Normal Address Space)

Figure 1-1. CP/M-8000 Default Memory Model

When a transient program is loaded it may be run in segmented or non-segmented mode and the address of the transient program (TPA) may be a segmented or non-segmented space. If the TPA is non-segmented it may combine or separate code and data, depending upon the linker options used to link the program and the space requirements of the transient program. Non-segmented transient programs may be run either in a single TPA segment space or in a segment space split into two spaces: one for instructions and one for data (called split I and D space). The Memory Region Table will decide in which physical segments to run the non-segmented program. If the program is to run with split instruction and data spaces, two physical segments are required (with the data, bss, and stack in the same physical segment), otherwise only a single physical segment is used. All addresses generated by a non-segmented transient program will have the same segment number.

A program running in segmented mode will be loaded into a segmented TPA in which addresses can have different segment numbers. This allows segmented programs to use any segment of the TPA, as specified in their object files. A segmented program requires the allocation of physical address segments to logical address segments, and this is accomplished during link time.

1.2 CP/M-8000 Organization

CP/M-8000 comprises three system modules: the Console Command Processor (CCP), the Basic Disk Operating System (BDOS), and the Basic Input/Output System (BIOS). These modules are linked together to form the operating system. They are discussed individually in this section.

1.3 Memory Layout

The CP/M-8000 operating system can reside anywhere in memory. The location of CP/M-8000 is defined during system generation. Typically the system occupies a segment which is logically separated from the TPA. See previous Figure 1-1 for an illustration of a logical segment separation of the system and the TPA.

The TPA for non-segmented programs consists of one to two 64 Kbyte segments, one for program code and one for data. Some programs expect program code and data to be mixed in one segment. The segment in which such programs are run may be the same as or different from the segments that contain programs with separated program code and data. The TPA for segmented programs consists of up to 128 segments.

The mapping of logical addresses (which consist of a 7-bit segment number and a 16-bit offset within a segment) into physical addresses is done by system-specific hardware, and the BIOS contains memory management operations to map addresses and copy blocks of memory. The two functions for map addressing and to copy blocks of memory are map_adr and mem_cpy. The function map_adr translates logical addresses into physical addresses and mem_cpy copies a specified number of bytes from one physical address to another.

See Figure 1-1 (CP/M-8000 Memory Model) for a discussion of the CP/M-8000 memory structure. In this memory model, the CP/M-8000 operating system resides in the System Memory and is the system address space (system operating mode). The system address space combines code and data, since there is no need for the extra space provided by code/data separation. The user task has a Transient Program Area (TPA) which resides in the normal address space (normal operating mode). The TPA may be segmented or non-segmented. The Base Page is in the highest part of data space and the user stack is just below it in data space.

As discussed earlier, memory locations are addressed by a seven-bit segment number and a sixteen-bit offset within the segment. This is not a linear but a two-dimensional space, capable of addressing 8 megabytes. Moreover, the System and Normal operating modes can have separate address spaces, so that a total of 16 megabytes of physical memory can be supported.

1.4 Console Command Processor (CCP)

The Console Command Processor, (CCP) provides the user interface to CP/M-8000. It uses the BDOS to read user commands and load programs, and provides several built-in user commands. It also provides parsing of command lines entered at the console.

1.5 Basic Disk Operating System (BDOS)

The Basic Disk Operating System (BDOS) provides operating system services to applications programs and to the CCP. These include character I/O, disk file I/O (the BDOS disk I/O operations comprise the CP/M-8000 file system), program loading, and others.

1.6 Basic I/O System (BIOS)

The Basic Input Output System (BIOS) is the interface between CP/M-8000 and its hardware environment. All physical input and output is done by the BIOS. It includes all physical device drivers, tables defining disk characteristics, and other hardware specific functions and tables. The CCP and BDOS do not change for different hardware environments because all hardware dependencies have been concentrated in the BIOS. Each hardware configuration needs its own BIOS. Section 4 describes the BIOS functions in detail. Section 5 discusses how to write a custom BIOS. A sample BIOS is presented in Appendix B.

1.7 I/O Devices

CP/M-8000 recognizes two basic types of I/O devices: character devices and disk drives. Character devices are devices that handle one character at a time. Disk devices handle data in units of 128 bytes, called sectors, and provide a large number of sectors which can be accessed in random, nonsequential, order. In fact, real systems might have devices with characteristics different from these. It is the BIOS's responsibility to resolve differences between the logical device models and the actual physical devices.

1.7.1 Character Devices

Character devices are input/output devices which accept or supply streams of ASCII characters to the computer. Typical character devices are consoles, printers, and modems. In CP/M-8000 operations on character devices are done one character at a time. A character input device sends ASCII CTRL-Z (1AH) to indicate end-of-file.

1.7.2 Disk Devices

Disk devices are used for file storage. They are organized into sectors and tracks. Each sector contains 128 bytes of data. If sector sizes other than 128 bytes are used on the actual disk, then the BIOS must do a logical-to-physical mapping to simulate 128-byte sectors to the rest of the system. All disk I/O in CP/M-8000 is done in one-sector units. A track is a group of sectors. The number of sectors on a track is a constant depending on the particular device. The characteristics of a disk device are specified in the Disk Parameter Block for that device. (See Section 5.)

To locate a particular sector, the disk, track number, and sector number must all be specified.

1.8 System Generation and Cold Start Operation

Generating a CP/M-8000 system is done by linking together the CCP, BDOS, and BIOS to create a file called CPM.SYS, which is the operating system. Section 2 discusses how to create CPM.SYS. CPM.SYS is brought into memory by a bootstrap loader, which typically resides on the first two tracks of a system disk. The term system disk as used here means a disk with the file CPM.SYS and a bootstrap loader, CPMLDR.SYS on the system tracks. Section 3 discusses the creation of a bootstrap loader.

End of Section 1

Section 2
System Generation

2.1 Overview

This section describes how to build a custom version of CP/M-8000 by combining your BIOS with the CCP and BDOS supplied by Digital Research to obtain a CP/M-8000 operating system suitable for your specific hardware system. Section 5 describes how to create a BIOS.

In this section, we assume that you have access to an already configured and executable CP/M-8000 system. If you do not, you should first read Section 6, which discusses how you can make your first CP/M-8000 system work.

A CP/M-8000 operating system is generated by using the linker, LD8K, to link together the system modules (CCP, BDOS, and BIOS) and bind the system to an absolute memory location. The resulting file is the configured operating system. It is named CPM.SYS.

2.2 Creating CPM.SYS

The CCP and BDOS for CP/M-8000 are distributed in a relocatable object code file named CPMSYS.REL. You must link your BIOS with CPMSYS.REL using the following command:

```
A>LD8K -W -O CPM.SYS BIOS.REL CPMSYS.REL -lCPM
```

where BIOS.REL is the compiled or assembled BIOS. This creates CPM.SYS, which is an absolute version of your system.

2.3 Relocating Utilities

Since the utilities all run in non-segmented mode, they do not need to be relocated: they will run in whatever segments you have assigned for the TPA. Note that the compiler and linker require separate code and data segments; all other utilities supplied with the system will run with nonsplit instruction and data segments when linked without the "-i" option of the linker.

End of Section 2

Section 3
Bootstrap Procedures

3.1 Bootstrapping Overview

Bootstrap loading is the process of bringing the CP/M-8000 operating system into memory and passing control to it. Bootstrap loading is necessarily hardware-dependent, and it is not possible to discuss all possible variations in this manual. However, the manual presents a model of bootstrapping that is applicable to many systems, and particularly to the Olivetti® M20™.

The model of bootstrapping that we present assumes that the CP/M-8000 operating system is to be loaded into memory from a disk in which the first few tracks (typically the first two) are reserved for the operating system bootstrap routines, while the remainder of the disk contains the file structure, consisting of a directory and disk files. (The topic of disk organization and parameters is discussed in Section 5.) In our model, the CP/M-8000 operating system resides in a disk file named CPM.SYS (described in Section 2), and the system tracks contain a bootstrap loader program (CPMLDR.SYS) that knows how to read CPM.SYS into memory and transfer control to it.

Most systems have a boot procedure similar to the following:

1. When you press reset, or execute a boot command from a monitor ROM, the hardware loads one or more sectors beginning at track 0, sector 1, into memory at a predetermined address, and then jumps to that address.

2. The code that came from track 0, sector 1, and is now executing, is typically a small bootstrap routine that loads the rest of the sectors on the system tracks (containing CPMLDR) into another predetermined address in memory, and then jumps to that address. Note that if your hardware is smart enough, steps 1 and 2 can be combined into one step.

3. The code loaded in step 2, which is now executing, is the CP/M Cold Boot Loader, CPMLDR, which is an abbreviated version of CP/M-8000 itself. CPMLDR now finds the file CPM.SYS, loads it, and jumps to it. A copy of CPM.SYS is now in memory, executing. This completes the bootstrapping process.

In order to create a CP/M-8000 diskette that can be booted, you need to know how to create CPM.SYS (see Section 2.2), how to create the Cold Boot Loader, CPMLDR, and how to put CPMLDR onto your system tracks. You must also understand your hardware enough to be able to design a method for bringing CPMLDR into memory and executing it.

3.2 Creating the Cold Boot Loader

CPMLDR is a miniature version of CPM.SYS. It contains stripped versions of the BDOS and BIOS, with only those functions which are needed to open the CPM.SYS file and read it into memory. CPMLDR exists in at least two forms; one form is the information in the system tracks, the other is a file named CPMLDR.SYS, which is created by the linker. The term CPMLDR is used to refer to either of these forms, but CPMLDR.SYS only refers to the file.

CPMLDR.SYS is generated using a procedure similar to that used in generating CPM.SYS. That is, a loader BIOS is linked with a loader system library, named CPMLDR.REL, to produce CPMLDR.SYS. Additional modules can be linked in as required by your hardware. The resulting file is then loaded onto the system tracks using the PUTBOOT utility program.

To perform the link and load, enter the following command line:

```
A> LD8K -W -O CPMLDR.SYS LDRBIOS.REL CPMLDR.REL -lCPM
```

3.2.1 Writing a Loader BIOS

The loader BIOS is very similar to your ordinary BIOS; it just has fewer functions, and the entry convention is slightly different. The following is a list of the differences.

1. Only one disk needs to be supported. The loader system selects only drive A. If you want to boot from a drive other than A, your loader BIOS should be written to select that other drive when it receives a request to select drive A.

2. The loader BIOS is not called through a trap; the loader BDOS calls an entry point named _bios instead. The parameters are still passed in registers, just as in the normal BIOS. Thus, your Function 0 does not need to initialize a trap, the code that in a normal BIOS is the Trap 3 handler should have the label _bios, and you exit from your loader BIOS with an RET instruction.

3. Only the following BIOS functions need to be implemented:

   0 (Init) Called just once, should initialize hardware as necessary, no return value necessary. Note that Function 0 is called via the _bios label with the function number equal to 0. You do not need a separate _init entry point.

   4 (Conout) Used to print error messages during boot. If you do not want error messages, this function should just be an RET instruction.

   9 (Seldsk) Called just once, to select drive A.

   10 (Settrk)

   11 (Setsec)

   12 (Setdma)

   13 (Read)

   16 (Sectran)

   18 (Get MRT) Not used now, but might be used in future releases.

   22 (Set exception)

4. You do not need to include an allocation vector or a check vector, and the Disk Parameter Header values that point to these can be anything. However, you still need a Disk Parameter Header, Disk Parameter Block, and directory buffer.

It is possible to use the same source code for both your normal BIOS and your loader BIOS if you use conditional compilation or assembly to distinguish the two. Appendix B provides an example of conditional compilation.

3.2.2 Building CPMLDR.SYS

Once you have written and compiled (or assembled) a loader BIOS, you can build CPMLDR.SYS in a manner very similar to building CPM.SYS. There is one additional complication here: the result of this step is placed on the system tracks. So, if you need a small prebooter to bring in the bulk of CPMLDR, the prebooter must also be included in the link you are about to do. The details of what must be done are hardware dependent, but the following example should help to clarify the concepts involved.

Suppose that your hardware reads track 0, sector 1, into memory at location <<B>>400H when reset is pressed, then jumps to 400H. Then your boot disk must have a small program in that sector that can load the rest of the system tracks into memory and execute the code that they contain. Suppose that you have written such a program, assembled it, and the assembler output is in BOOT.O. Also assume that your loader BIOS object code is in the file LDRBIOS.REL. Then the following command links together the code that must go on the system tracks.

```
A>LD8K -W -O CPMLDR.SYS BOOT.O LDRBIOS.REL CPMLDR.REL -/CPM
```

Once you have created CPMLDR.SYS in this way, you can use the PUTBOOT utility to place it on the system tracks. PUTBOOT is described in Section 8. The command to place CPMLDR on the system tracks of drive A is

```
A>PUTBOOT CPMLDR.SYS A:
```

PUTBOOT reads the file CPMLDR.SYS and puts the result on the specified drive. After you have copied CPM.SYS to the disk, you can boot from it.

3.3 Introduction to the CP/M-8000 Target Machine

This section presents the Olivetti M20 as a specific microcomputer model chosen to implement the CP/M-8000 operating system. The difference between the M20 model and the generic model is the bootstrapping loader placement.

The Olivetti M20 uses the Zilog Z8001 microprocesor and is capable of supporting up to 512K bytes of physical memory. The standard configuration of the M20 includes a monochrome, bit-mapped display screen and two built-in 5 1/4 inch floppy disk drives with a capacity of 320K bytes each. Optionally, one of the floppy disk drives can be replaced by a built-in hard disk drive with 8 megabytes of storage. The M20 also has a serial port, which may be attached to a line printer or to another computer.

3.3.1 M20 Memory Architecture

When implementing or porting over the CP/M-8000 operating system to the Olivetti M20, the memory architecture of the M20 requires particular attention. Physical memory in the M20 is configured in banks of 16K bytes each. The mapping between segmented addresses and physical memory banks is done through a ROM, and thus is not programmable by the user. The M20 contains a memory map for a configuration with 256K bytes and a monochrome display. One 16K bank is set aside for the bit-mapped display. This memory is addressed as segment 3. Segment 4 addresses the bootstrap ROM, and segment 2 is used as RAM by the ROM. The rest of memory is available for program use. Note that the same banks of physical memory can be addressed in various ways. For instance, segment 8 has a separated code and data space. Segment 10 has a combined code and data space, whose physical memory is the same as the segment 8 code space physical memory.

3.3.2 CP/M-8000 Implementation

In this implementation, the CP/M-8000 operating system resides in segment 11. Non-segmented user programs are loaded into segment 8 if they require separate code and data, or into segment 10 if they use combined code and data space. The information describing the type of space a program needs is present in the first word of the program's object file on disk. Segmented programs are loaded into whatever segment numbers are in their object file.

3.3.3 Display and Disk Drivers

In the M20 bootstrap ROM are drivers for the display screen and the disks. The user may write his own drivers or use the drivers provided here. The CP/M-8000 BIOSIO module invokes the segmented addressing mode after which all of memory is addressable. The BIOSIO module then calls the ROM drivers indirectly through a branch table which resides at a fixed location in the ROM. This feature makes it unnecessary to change the CP/M-8000 BIOSIO module to operate compatibly with any future versions of the Olivetti ROM.

3.3.4 Addressing the Screen Bit Map

Graphics programs may be implemented easily using the SC #1 memory management primitives. The user program builds an image of the bit map in a 16K buffer of its own. The user program then calls mem_cpy to copy that buffer to segment 3, locations 0 through 16383. Alternatively, a user program can invoke the segmented addressing mode by executing a _map_adr system call. The program then will address the bit map memory directly. The programmer should be aware that entering segmented mode has side effects for which the user program must compensate.

End of Section 3

Section 4
BIOS Functions

4.1 Introduction

All CP/M-8000 hardware dependencies are concentrated in subroutines that are collectively referred to as the Basic I/O System (BIOS). A CP/M-8000 system implementor can tailor CP/M-8000 to fit nearly any Z8000 operating environment. This section describes each BIOS function: its calling conventions, parameters, and the actions it must perform. The discussion of Disk Definition Tables is treated separately in Section 5.

When the BDOS calls a BIOS function, it places the function number in register R3, and function parameters in registers RR4 and RR6. It then executes a SC #3 instruction. R3 is always needed to specify the function, but each function has its own requirements for other parameters. Specific parameter requirements are provided in the description of each function. The BIOS uses RR6 to return any values to the caller. The size of the returned value depends on the particular BIOS function. Byte values contained in word or long-word length registers are null padded.

Note: The system call handler in the BIOS must preserve at least registers R8 through R15. The handlers provided in most BIOS's preserve all registers, except for RR6 which is used to return results. Of course, if the BIOS uses interrupts to service I/O, the interrupt handlers will need to preserve registers.

Table 4-1 summarizes BIOS register usage.

User applications typically do not need to make direct use of BIOS functions. However, when access to the BIOS is required by user software, it should use the BDOS Direct BIOS Function, Call 50, instead of calling the BIOS with a SC #3 instruction. This rule ensures that applications remain compatible with future systems.

The BIOS must also maintain a vector of Exception Handler addresses, through which all system calls and traps are routed. The vector numbers have been selected to match the exception used in CP/M-68K. These numbers will be found in the Programmer's Guide.

Section 4.2 describes the system calls for Z8000 memory management.

The Disk Parameter Header (DPH) and Disk Parameter Block (DPB) formats have changed slightly from previous CP/M versions to accommodate the Z8000's 32-bit addresses. The formats are described in Section 5.

Table 4-1. BIOS Register Usage

Entry Parameters:

R3 = function code
RR4 = first parameter
RR6 = second parameter

Return Values:

RL7 = byte values (8 bits)
R7 = word values (16 bits)
RR6 = longword values (32 bits)

The decimal BIOS function numbers and the functions they correspond to are listed in Table 4-2.

Table 4-2. BIOS Functions

Number	Function
0	Initialization (called for cold boot)
1	Warm Boot (called for warm start)
2	Console Status (check for console character ready)
3	Read Console Character In
4	Write Console Character Out
5	List (write listing character out)
6	Auxiliary Output (write character to auxiliary output device)
7	Auxiliary Input (read from auxiliary input)
8	Home (move to track 00)
9	Select Disk Drive
10	Set Track Number
11	Set Sector Number

Table 4-2. (continued)

Number	Function
12	Set DMA Address
13	Read Selected Sector
14	Write Selected Sector
15	Return List Status
16	Sector Translate
18	Get Memory Region Table Address
19	Get I/O Mapping Byte
20	Set I/O Mapping Byte
21	Flush Buffers
22	Set Exception Handler Address

FUNCTION 0: INITIALIZATION

Entry Parameters:
Register R3: 00H

Returned Value:
Register R7: User/Disk Numbers

BIOS Function 0 executes the cold bootstrap sequence and initializes the BIOS. Unlike other BIOS functions, this function is not invoked with an SC #3 instruction. Instead, a jump to the "entry:" label in the biosboot module invokes this function to execute. The biosboot module sets up the PSA system segment and system stack pointer, then jumps to the to a location labeled "bios" to invoke this function.

Function 0 calls _trapinit and _biosinit to enable the BIOS. The _trapinit routine initializes the trap handler table. The _biosinit routine initializes the hardware and internal BIOS variables. Function 0 then transfers control to the CCP.

Function 0 returns a longword value. The CCP uses this value to set the initial user number and the initial default disk drive. The least significant byte of RR6 is the disk number (0 for drive A, 1 for drive B, and so on). The next most significant byte is the user number. The high-order bytes should be zero.

The entry point to this function must be named bios and must be declared global. This function is called only once from the system at system initialization.

For an example of bootstrap code, see the BIOSBOOT.8KN file on the distribution disk.

FUNCTION 1: WARM BOOT

Entry Parameters:
Register R3: 01H

Returned Value: None

This function is called whenever a program terminates. Some reinitialization of the hardware or software might occur. When this function completes, it jumps directly to the entry point of the CCP, named _ccp. Note that _ccp must be declared as a global.

FUNCTION 2: CONSOLE STATUS

```
Entry Parameters:
     Register R3:  02H

  Returned Value:
     Register R7:  00FFH if ready
     Register R7:  0000H if not ready
```

This function returns the status of the currently assigned console device. It returns 00FFH in register R7 when a character is ready to be read, or 0000H in register R7 when no console characters are ready.

FUNCTION 3: READ CONSOLE CHARACTER

Entry Parameters:
Register R3: 03H

Returned Value:
Register R7: Character

This function reads the next console character into register R7. If no console character is ready, it waits until a character is typed before returning.

FUNCTION 4: WRITE CONSOLE CHARACTER

Entry Parameters:
Register R3: 04H
Register R5: Character

Returned Value: None

This function sends the character from register R5 to the console output device. The character is in ASCII. You might want to include a delay or filler characters for a line-feed or carriage return, if your console device requires some time interval at the end of the line (such as a TI Silent 700 Terminal™). You can also filter out control characters that have undesirable effects on the console device.

FUNCTION 5: LIST CHARACTER OUTPUT

Entry Parameters:
Register R3: 05H
Register R5: Character

Returned Value: None

This function sends an ASCII character from register R5 to the currently assigned listing device. If your list device requires some communication protocol, it must be handled here.

FUNCTION 6: AUXILIARY OUTPUT

Entry Parameters:
Register R3: 06H
Register R5: Character

Returned Value: None

This function sends an ASCII character from register R5 to the currently assigned auxiliary output device.

FUNCTION 7: AUXILIARY INPUT

Entry Parameters:
Register R3: 07H

Returned Value:
Register R7: Character

This function reads the next character from the currently assigned auxiliary input device into register R7. It reports an end-of-file condition by returning an ASCII CTRL-Z (1AH).

FUNCTION 8: HOME

Entry Parameters:
Register R3: 08H

Returned Value: None

This function returns the disk head of the currently selected disk to the track 00 position. If your controller does not have a special feature for finding track 00, you can translate the call to a SETTRK function with a parameter of 0.

FUNCTION 9: SELECT DISK DRIVE

Entry Parameters:
Register R3: 09H
Register R5: Disk Drive
Register R7: Logged-in Flag

Returned Value:
Register RR6: Address of Selected Drive's DPH

This function selects the disk drive specified in register R5 for further operations. Register R5 contains 0 for drive A, 1 for drive B, up to 15 for drive P.

On each disk select, this function returns the address of the selected drive's Disk Parameter Header in register RR6. See Section 5 for a discussion of the Disk Parameter Header.

This function must return 00000000H in register RR6 if a nonexistent drive has been indicated in register R5. Although the function must return the header address on each call, it is advisable to postpone the actual physical disk select operation until an I/O function (seek, read, or write) is performed. Disk select operations can occur without a subsequent disk operation. Thus, doing a physical select each time this function is called may waste time.

If the least significant bit in register R7 is zero on entry to the Select Disk Drive function, the disk is not currently logged in. If the disk drive is capable of handling varying media (such as single- and double-sided, single- and double-density disks), the BIOS should check the type of media currently installed and then set up the Disk Parameter Block.

FUNCTION 10: SET TRACK NUMBER

Entry Parameters:
    Register R3: 0AH
    Register R5: Disk track number

Returned Value: None

This function specifies in register R5 the disk track number for use in subsequent disk accesses. The track number remains active until either another Function 10 or a Function 8 (Home) is performed.

You can choose to physically seek to the selected track at this time, or delay the physical seek until the next read or write actually occurs.

The track number can range from 0 to the maximum track number supported by the physical drive. However, the maximum track number is limited to 65535 by the fact that it is being passed as a 16-bit quantity. Standard floppy disks have tracks numbered from 0 to 76.

FUNCTION 11: SET SECTOR NUMBER

Entry Parameters:
Register R3: 0BH
Register R5: Sector Number

Returned Value: None

This function specifies in register R5 the sector number for subsequent disk accesses. This number remains active until Function 11 is called again.

The function selects actual (unskewed) sector numbers. If skewing is appropriate, call Function 16 previous to calling Function 11.

You can send the sector number information to the controller after executing Function 11, or you may delay sector selection until a read or write operation occurs.

FUNCTION 12: SET DMA ADDRESS

Entry Parameters:
Register R3: 0CH
Register RR4: DMA Address

Returned Value: None

This function contains the DMA (disk memory access) address in register RR4 for subsequent read or write operations. Note that the controller need not actually support direct memory access. The BIOS uses the 128-byte area starting at the selected DMA address for the memory buffer during the following read or write operations. This function can be called with either an even or an odd address for a DMA buffer.

FUNCTION 13: READ SECTOR

Entry Parameters:
Register R3: 0DH

Returned Value:
Register R7: 0 if no error
Register R7: 1 if physical error

After the drive has been selected, the track has been set, the sector has been set, and the DMA address has been specified, the Read Sector Function uses these parameters to read one sector and returns the error code in register R7.

Currently, CP/M-8000 responds only to a zero or nonzero return code value. If the value in register R7 is zero, CP/M-8000 assumes that the disk operation completed properly. If an error occurs, the BIOS should attempt at least ten retries to see if the error is recoverable.

FUNCTION 14: WRITE SECTOR

Entry Parameters:
Register R3: 0EH
Register R5: 0=normal write
1=write to a directory sector
2=write to first sector of new block

Returned Value:
Register R7: 0=no error
1=physical error

This function is used to write 128-byte data blocks from the currently selected DMA buffer to the currently selected sector, track, and disk. The value in register R5 indicates whether the write is an ordinary write operation or whether there are special considerations.

If register R5=0, this is an ordinary write operation. If R5=1, this is a write to a directory sector, and the write should be physically completed immediately. If R5=2, this is a write to the first sector of a newly allocated block of the disk. The significance of this value is discussed in Section 5 under Disk Buffering.

FUNCTION 15: RETURN LIST STATUS

Entry Parameters:
    Register R3: 0FH

Returned Value:
    Register R7: 00FFH=device ready
    Register R7: 0000H=device not ready

This function returns the status of the list device. Register R7 can contain 0000H to indicate that the list device is not ready to accept a character, or 00FFH to indicate that the list device is ready.

FUNCTION 16: SECTOR TRANSLATE

Entry Parameters:
Register R3: 10H
Register R5: Logical Sector Number
Register RR6: Address of Translate Table

Returned Value:
Register R7: Physical Sector Number

This function performs logical-to-physical sector translation, as discussed in Section 5.2.2. The Sector Translate function receives a logical sector number from register R5. The logical sector number can range from 0 to the number of sectors per track minus one. Function 16 also receives the address of the translate table in register RR6. This address must be in the system's address space. The logical sector number is used as an index into the translate table. The resulting physical sector number is returned in R7.

If register RR6 = 00000000H, indicating that there is no translate table, register R5 is copied to register R7 before Function 16 returns. Note that other algorithms are possible; in particular, it is common to increment the logical sector number in order to convert the logical sector range of 0 to n-1 into the physical range of 1 to n. Sector Translate is always called by the BDOS, whether the translate table address in the Disk Parameter Header is zero or nonzero.

FUNCTION 18: GET ADDRESS OF MEMORY REGION TABLE

Entry Parameters:
Register R3: 12H

Returned Value:
Register RR6: Memory Region Table Address

This function returns the address of the Memory Region Table (MRT) in register RR6. The MRT, which must be present and must begin on an even address, describes the segments that compose the TPA for non-segmented programs. The format of the MRT is shown below:

Entry Count (always = 4)	16 bits
Base address of first region	32 bits
Length of first region	32 bits
Base address of second region	32 bits
Length of second region	32 bits
Base address of third region	32 bits
Length of third region	32 bits
Base address of fourth region	32 bits
Length of fourth region	32 bits

Figure 4-1. Memory Region Table Format

The regions are:

Region 1 - the segment used for programs with merged program and data segments;

Region 2 - the instruction segment for programs with split instruction and data segments;

Region 3 - the data segment for programs with split instruction and data segments;

Region 4 - an instruction segment from which a program can access the instructions in region 2 as data.

A program with instructions residing in region 4 can access the instructions stored in region 2 as data. The segment number field of the program counter of a such a program in region 4 can be the segment number of region 2.

FUNCTION 19: GET I/O BYTE

Entry Parameters:
Register R3: 13H

Returned Value:
Register R7: I/O Byte Current Value

This function returns the current value of the logical to physical input/output device byte (I/O byte) in register R7. This 8-bit value associates physical devices with CP/M-8000's four logical devices as noted in Figure 4-2. Table 4-3 defines these devices. Note that even though this is a byte value, we are using word references. The upper byte must be zero.

The I/O byte is split into four 2-bit fields called CONSOLE, AUXILIARY INPUT, AUXILIARY OUTPUT, and LIST, as shown in Figure 4-2.

Most Significant Least Significant

I/O Byte	LIST	AUXILIARY OUTPUT	AUXILIARY INPUT	CONSOLE
bits:	7,6	5,4	3,2	1,0

Figure 4-2. I/O Byte Fields

Peripheral devices other than disks are seen by CP/M-8000 as logical devices, and are assigned to physical devices within the BIOS. Device characteristics are defined in Table 4-3.

Table 4-3. CP/M-8000 Logical Device Characteristics

Device Name	Characteristics
CONSOLE	The interactive console that you use to communicate with the system is accessed through functions 2, 3 and 4. Typically, the console is a CRT or other terminal device.
LIST	The listing device, usually a printer.
AUXILIARY OUTPUT	An optional serial output device.
AUXILIARY INPUT	An optional serial input device.

The value in each I/O Byte field can be in the range 0-3, defining the assigned source or destination of each logical device. The values that can be assigned to each field are given in Table 4-4.

Note that a single peripheral can be assigned as the LIST, AUXILIARY INPUT, and AUXILIARY OUTPUT device, simultaneously. If no peripheral devices are assigned to LIST, AUXILIARY INPUT, or AUXILIARY OUTPUT, your BIOS should give an appropriate error message. This prevents system hang-up if the device is accessed by PIP or some other transient program. Alternatively, the AUXILIARY OUTPUT and LIST functions simply can return to the caller, and the AUXILIARY INPUT function can return with a 1AH (CTRL-Z) in register R7 to indicate an immediate end-of-file.

Table 4-4. I/O Byte Field Definitions

Bit	Definition
CONSOLE field (bits 1,0)	
0	console is assigned to the console printer (TTY:)
1	console is assigned to the CRT device (CRT:)
2	batch mode: use the AUXILIARY INPUT as the CONSOLE input, and the LIST device as the CONSOLE output (BAT:)
3	user defined console device (UC1:)
AUXILIARY INPUT field (bits 3,2)	
0	AUXILIARY INPUT is the Teletype device (TTY:)
1	AUXILIARY INPUT is the high-speed reader device (PTR:)
2	user defined reader #1 (UR1:)
3	user defined reader #2 (UR2:)
AUXILIARY OUTPUT field (bits 5,4)	
0	AUXILIARY OUTPUT is the Teletype device (TTY:)
1	AUXILIARY OUTPUT is the high-speed punch device (PTP:)
2	user defined punch #1 (UP1:)
3	user defined punch #2 (UP2:)
LIST field (bits 7,6)	
0	LIST is the Teletype device (TTY:)
1	LIST is the CRT device (CRT:)
2	LIST is the line printer device (LPT:)
3	user defined list device (UL1:)

The implementation of the I/O byte is optional, and affects only the organization of your BIOS. The only CP/M-8000 utilities to use the I/O byte are PIP and STAT. PIP allows access to the physical devices. STAT allows logical-physical assignments to be made and displayed. It is good practice first to implement and test your BIOS without the IOBYTE functions, then to add the I/O byte function after testing.

FUNCTION 20: SET I/O BYTE

Entry Parameters:
    Register R3: 14H
    Register R5: Desired

Returned Value: None

This function uses the value in register R5 to set the value of the I/O Byte. See Table 4-4 for the I/O byte field definitions. Because this is a byte value, the most significant byte must be zero.

FUNCTION 21: FLUSH BUFFERS

Entry Parameters:
Register R3: 15H

Returned Value:
Register R7: 0000H=successful write
Register R7: FFFFH=unsuccessful write

This function forces the contents of any disk buffers that have been modified to be written. After this function has been performed, all disk writes have been physically completed. After the buffers are written, this function returns a zero in register R7. However, if the buffers cannot be written or an error occurs, the function returns a value of FFFFH in register R7.

FUNCTION 22: SET EXCEPTION HANDLER ADDRESS

Entry Parameters:
Register R3: 16H
Register R5: Exception Vector Number
Register RR6: Exception Vector Address

Returned Value:
Register RR6: Previous Vector Contents

This function sets the exception vector indicated in register R5 to the value specified in register RR6. The previous vector value is returned in register RR6. Unlike the BDOS Set Exception Vector Function (61), this BIOS function sets any exception vector. Note that register R5 contains the exception vector number. Thus, to set exception #2, segmentation trap, this register contains a 2.

The exception handler is called as a subroutine, with all of its registers saved on the stack, in the form given for the context block in the Transfer Control instruction. On a segmented CPU, the exception handler is entered in segmented mode. It should return with a RET instruction.

All of the caller's registers except RR0 are also passed intact to the handler.

4.2 Memory Management System Calls

The system call SC #1 is used for memory management operations: mapping addresses from logical to physical, copying blocks of (physical) memory, and transferring control from one address space to another. Parameters are specified in registers RR2, RR4, and RR6, and a value may be returned in RR6. The SC #1 descriptions below illustrate the register settings to use when making the calls with assembly language as well as the C language calling sequence. The C language library contains system call SC #1 functions designed to be called from non-segmented C programs.

To use the memory management system calls successfully, take care to distinguish between logical and physical addresses. A logical address refers to an address in a program's address space; it is 16 bits long for a non-segmented program, and 23 bits long (stored in a 32-bit word) for a segmented program. A physical address is the address of the physical memory which the processor accesses. Two cases illustrate the necessity of this distinction: first, the hardware may map a logical address to derive from it a physical address. Second, a default segment number associates with the logical addresses of a non-segmented program running on a segmented CPU. This default segment number is taken from the program counter (PC).

For CPM-8000, it is necessary that the logical-to-physical mapping process not affect the low-order 16 bits (offset part) of an address. Thus, on systems with MMU's that permit segments to start on arbitrary boundaries, the apparently "physical" addresses used in the BIOS code might be subject to further mapping by the MMU. So, when writing a BIOS for such systems it is necessary to distinguish the memory segments which belong to the system addresses from those which belong to the TPA.

BIOS operations done through BDOS call 50 are mapped from the caller's address space into physical addresses.

SYSTEM CALL 1: MEMORY COPY

Entry Parameters:
Register RR2: Length
Register RR4: Destination
Register RR6: Source

Returned Value: None

C language call sequence:

```
long source, dest, length;

/* source: source address.
   dest: destination address.
   length: length of block in bytes. */

 mem_cpy ( source, dest, length )
```

This operation copies a block of Length bytes from Source to Destination. Length must be greater than zero and less than 65536 (a Length of zero is used to distinguish different memory management operations). The Source and Destination are segmented physical addresses, as provided by the Map Address operation below.

SYSTEM CALL 1: MAP ADDRESS

Entry Parameters:
Register RR2: 0
Register RR4: Space Code
Register RR6: Logical Address

Returned Value:
Register RR6: Physical Address

C language call sequence:

```
long addr, paddr; int space;

/* addr: Logical Address.
   paddr: Physical Address
          (returned value)
   space: Space Code. */

paddr =  map_adr ( addr, space )
```

This form of SC #1 is used to convert a logical address to a physical address. Since logical addresses depend on both the mode (system or normal) of the program using them, and on the space being accessed (program or data), a code determines from which space to map.

If the program in the TPA is running non-segmented, the Set TPA Segment version of SC #1 will have been used to tell the mapping routine which segment is being used. If the TPA is running with split program and data, it is also necessary to distinguish between the segment number that goes in the program counter to access instructions, and the physical segment by which the TPA's instruction segment can be accessed as data.

The space codes are as follows:

0: Caller's Data Space
257: Caller's Program Space (as Instructions)
2: System's Data Space
3: System's Program Space (as Data)
259: System's Program Space (as Instructions)
4: TPA's Data Space
5: TPA's Program Space
261: TPA's Program Space (as Instructions)

SYSTEM CALL 1: SET TPA SEGMENT

Entry Parameters:
Register RR2: 0
Register RR4: 0000FFFFh
Register RR6: TPA Base Address

Returned Value: None

C language call sequence:

(This function uses the map_adr function with special parameter values.)

```
long addr, paddr;
/* addr: TPA Base Address */

 map_adr ( addr, -1 )
```

This operation sets the base segment for a non-segmented program running in the TPA. This base address is usually obtained from entry 1 in the Memory Region Table for programs with instructions and data in the same segment, and from entry 2 for programs with split instruction and data segments.

If R6 (the high-order word of RR6) is FFFFh, the program running in the TPA is assumed to be running in segmented mode.

SYSTEM CALL 1: TRANSFER CONTROL

Entry Parameters:
Register RR2: 0
Register RR4: FFFEh
Register RR6: Context Block Address

Returned Value: none

C language call sequence:

```
long context;

/* context: Context Block Address. */

xfer (&context )
```

This operation causes control to be transferred to another address space. It allows all of the registers to be specified except for the system mode stack pointer. DDT™ uses this operation to transfer control to the program being debugged. RR6 points to a context block of the form:

```
word     R0
word     R1
word     R2
word     R3
word     R4
word     R5
word     R6
word     R7
word     R8
word     R9
word     R10
word     R11
word     R12
word     R13
word     R14 (normal mode R14)
word     R15 (normal mode R14)
word     ignored
word     FCW (Flag/Control Word)
word     PC Segment
word     PC Offset
```

Note that the PC segment word is required for compatibility even if the CPU is a non-segmented Z8002.

End of Section 4

Section 5
Creating a BIOS

5.1 Overview

The BIOS provides a standard interface to the physical input/output devices in your system. The BIOS interface is defined by the functions described in Section 4. Those functions, taken together, constitute a model of the hardware environment. Each BIOS is responsible for mapping that model onto the real hardware.

In addition, the BIOS contains disk definition tables that define the characteristics of the disk devices that are present, and provides some storage for use by the BDOS maintaining disk directory information.

Section 4 describes the functions that must be performed by the BIOS, and the external interface to those functions. This Section contains additional information describing the structure and significance of the disk definition tables and information about sector blocking and deblocking. Careful choices of disk parameters and disk buffering methods are necessary if you are to achieve the best possible performance from CP/M-8000. Therefore, you should read this section thoroughly before writing a custom BIOS.

5.2 Disk Definition Tables

As in other CP/M systems, CP/M-8000 defines disk device charateristics through a set of tables. This section describes each table and discusses parameter options.

5.2.1 Disk Parameter Header

Each disk drive has an associated 26-byte Disk Parameter Header (DPH) which both contains information about the disk drive and provides a scratchpad area for certain BDOS operations. Each drive must have its own unique DPH. The format of a Disk Parameter Header is shown in Figure 5-1.

XLT	0000	0000	0000	DIRBUF	DPB	CSV	ALV
32b	16b	16b	16b	32b	32b	32b	32b

Figure 5-1. Disk Parameter Header

Each element of the DPH is either a word or longword value. Table 5-1 gives the meanings of the Disk Parameter Header (DPH) elements.

Table 5-1. Disk Parameter Header Elements

Element	Description
XLT	Address of the logical-to-physical sector translation table. If there is no translation table, it contains the value 0, and the physical and logical sector numbers will be identical. Disk drives with identical sector translation can share the same translate table. Section 5.2.2 describes the sector translation table.
0000	Three scratchpad words for use within the BDOS.
DIRBUF	Address of a 128-byte scratchpad area for directory operations within BDOS. All DPHs address the same scratch pad area.
DPB	Address of a disk parameter block for this drive. Drives with identical disk characteristics can address the same disk parameter block.

Table 5-1. (continued)

Element	Description
CSV	Address of a checksum vector. The BDOS uses this area to maintain a vector of directory checksums for the disk. These checksums detect when the disk in a drive has been changed. If the disk is not removable, then it is not necessary to have a checksum vector. Each DPH must point to a unique checksum vector. The checksum vector should contain 1 byte for every four directory entries, or 128 bytes of directory. The length of the checksum vector is equal to (DRM+1) / 4. Section 5.2.3 discusses the DRM value.
ALV	Address of the allocation vector, a scratchpad area used by the BDOS to keep disk storage allocation information. The area must be unique for each DPH. There must be one bit for each allocation block on the drive. This requires that the length of the allocation vector be equal to (DSM/8) + 1. Section 5.2.3 discusses the DSM value.

5.2.2 Sector Translate Table

Sector translation in CP/M-8000 is a method of logically renumbering the sectors on each disk track to improve disk I/O performance. Frequently programs must access disk sectors sequentially. However, in reading sectors sequentially, most programs lose a full disk revolution between sectors because there is not enough time between adjacent sectors to begin a new disk operation. To alleviate this problem, the traditional CP/M solution is to create a logical sector numbering scheme in which logically sequential sectors are physically separated. Thus, between two logically contiguous sectors, there is a rotational delay. The sector translate table defines the logical-to-physical mapping for a particular drive.

Sector translate tables are used only within the BIOS, and may have any convenient format. The only interaction the BDOS has with the table is to fetch the sector translate table address from the DPH and to pass that address to the Sector Translate Function of the BIOS. The most common form for a sector translate table is an n-byte or n-word array of physical sector numbers, where n is the number of sectors per disk track. Indexing into the table with the logical sector number yields the corresponding physical sector number.

Although you may choose any convenient logical-to-physical mapping, there is a nearly universal mapping used in the CP/M community for single-sided, single-density, 8-inch diskettes. That mapping is shown in Figure 5-2. Your choice of mapping affects diskette compatibility among different systems. To make your mapping compatible with different systems, we recommend the mapping shown in Figure 5-2.

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12
Physical Sector	1	7	13	19	25	5	11	17	23	3	9	15	21
Logical Sector	13	14	15	16	17	18	19	20	21	22	23	24	25
Physical Sector	2	8	14	20	26	6	12	18	24	4	10	16	22

Figure 5-2. Sample Sector Translate Table

5.2.3 Disk Parameter Block

A Disk Parameter Block (DPB) defines several characteristics associated with a particular disk drive. These include the size of the drive, the number of sectors per track, and the amount of directory space.

One or more DPH's may use a common DPB if the disks are identical in definition. Figure 5-3 shows the DPB format. Table 5-2 describes the DPB fields.

SPT	BSH	BLM	EXM	0	DSM	DRM	Reserved	CKS	OFF
16b	8b	8b	8b	8b	16b	16b	16b	16b	16b

Figure 5-3. Disk Parameter Block

Each field is a word or a byte value. Table 5-2 describes each field.

Table 5-2. Disk Parameter Block Fields

Field	Definition
SPT	Number of 128-byte logical sectors per track.
BSH	The block shift factor, determined by the data block allocation size, as shown in Table 5-3.
BLM	The block mask, determined by the data block allocation size, as shown in Table 5-3.
EXM	The extent mask, determined by the data block allocation size and the number of disk blocks, as shown in Table 5-4.
0	Reserved byte.
DSM	Determines the total storage capacity of the disk drive and is the number of the last block, zero relative. The disk contains DSM+1 blocks.
DRM	Determines the total number of directory entries that can be stored on this drive. DRM is the number of the last directory entry, zero relative. The disk contains DRM+1 directory entries. Each directory entry requires 32 bytes. For maximum efficiency the value of DRM should be such that the directory entries exactly fill an integral number of allocation units.
CKS	The size of the directory check vector. The CKS value is zero if the disk is permanently mounted. The CKS value is equal to (DRM) / 4 + 1 for removable media.
OFF	The number of reserved tracks at the beginning of a logical disk. This is the number of the track on which the directory begins.

In order to select appropriate values for the Disk Parameter Block elements, you must understand how disk space is organized in CP/M-8000. A CP/M-8000 disk has two major areas: the boot or system tracks, and the file system tracks. The boot tracks hold a machine-dependent bootstrap loader for the operating system. They consist of tracks 0 to OFF-1. Zero is a legal value for OFF, and in that case, there are no boot tracks. The usual value of OFF for 8-inch floppy disks is two.

The tracks after the boot tracks, beginning with track number OFF, contain the disk directory and disk files. Disk space in this area is grouped into units called allocation units or blocks. The block size for a particular disk is a constant, called BLS.

BLS can take on any one of these values: 1024, 2048, 4096, 8192, or 16384 bytes. No other values for BLS are allowed. Note that BLS does not appear explicitly in any BIOS table. However, it determines the values of a number of other parameters. The DSM field in the Disk Parameter Block is one less than the number of blocks on the disk. Space is allocated to a file or to the directory in whole blocks. No fraction of a block can be allocated.

The choice of BLS is very important. It affects the efficient use of disk space. There is a minimum value of BLS that allows an entire disk to be used. Each block on the disk has a block number from 0 to DSM. The largest block number allowed is 32767. Therefore, the largest number of bytes that can be addressed in the file system space is 32768 * BLS. Because the largest allowable value for BLS is 16384, the disk capacity that CP/M-800 can access is 16384*32768 = 512 Mbytes.

Each directory entry can contain either 8 block numbers, if DSM is greater than or equal to 256, or 16 block numbers if DSM is less than 256. Each file needs sufficient directory entries to hold the block numbers of all blocks allocated to the file. A large value for BLS implies that fewer directory entries are needed. If fewer directory entries are used, directory search time is decreased.

The disadvantage of a large value for BLS is that files are allocated BLS bytes at a time, and there is potentially a large unused portion of a block at the end of the file. If there are many small files on a disk, the waste can be significant.

The BSH and BLM parameters in the DPB are functions of BLS. Once you have chosen BLS, use Table 5-3 to determine BSH and BLM. The EXM parameter of the DPB is a function of BLS and DSM. Use Table 5-4 to find the value of EXM for your disk.

Table 5-3. BSH and BLM Values

BLS	BSH	BLM
1024	3	7
2048	4	15
4096	5	31
8192	6	63
16384	7	127

Table 5-4. EXM Values

BLS	DSM <= 255	DSM > 255
1024	0	N/A
2048	1	0
4096	3	1
8192	7	3
16384	15	7

The DRM entry in the DPB is one less than the total number of directory entries. Choose a DRM value large enough so that you do not run out of directory entries before running out of disk space. It is not possible to give an exact rule for determining DRM because the number of directory entries needed depends on the number and sizes of the files present on the disk.

The CKS entry in the DPB is the byte count of the checksum vector. The CSV field of the DPH points to the checksum vector. If the disk is not removable, a checksum vector is not needed, so this value can be zero.

5.3 Disk Blocking

When the BDOS performs a disk read or write operation using the BIOS, the unit of information read or written is a 128-byte sector. This might correspond to the actual physical sector size of the disk. If not, the BIOS must implement a method of representing the 128-byte sectors used by CP/M-8000 on the actual device. Usually if the physical sectors are not 128 bytes long, they are some multiple of 128 bytes. Thus, one physical sector can hold some integer number of 128-byte CP/M sectors. In this case, any disk I/O actually transfers several CP/M sectors at once.

It might also be desirable to perform disk I/O in units of several 128-byte sectors to increase disk throughput by decreasing rotational latency. Rotational latency is the average time it takes for the desired position on a disk to rotate around to the read-write head. Generally this averages 1/2 disk revolution per transfer. Because much disk I/O is sequential, rotational latency can be greatly reduced by reading several sectors at a time, and storing them for future use.

In both the preceding cases, the point of interest is that physical I/O occurs in units larger than 128-byte sectors. Section 5.3.1 discusses methods of performing disk I/O in units larger than 128-byte sectors.

5.3.1 A Simple Approach

This section presents a simple approach to handling a physical sector size larger than the 128-byte logical sector size. The method discussed in this section is a starting point for refinements discussed in the following sections. Its simplicity makes it a logical choice for a first BIOS on new hardware. However, the disk throughput that you can achieve with this method is poor, and the refinements discussed later give dramatic improvements.

Probably the easiest method for handling a physical sector size that is a multiple of 128 bytes is to have a single buffer the size of the physical sector internal to the BIOS. Then, when a disk read occurs the physical sector containing the desired 128- byte logical sector is read into the buffer, and the appropriate 128 bytes are copied to the DMA address. Writing is a little more complicated: you must put data into a 128-byte portion of the physical sector, but you can only write a whole physical sector. Therefore, you must first read the physical sector into the BIOS's buffer, copy the 128 bytes of output data into the proper 128-byte piece of the physical sector in the buffer, and finally, write the entire physical sector back to disk.

Note: This operation involves two rotational latency delays in addition to the time needed to copy the 128 bytes of data. In fact, the second rotational wait is probably nearly a full disk revolution, since the copying is usually much faster than a disk revolution.

5.3.2 Some Refinements

There are many methods you may use to improve the performance of the algorithm of Section 5.3.1. The first method is based on the fact that disk accesses are usually done sequentially. Thus, if data from a certain physical sector is needed, it is likely that another piece of that sector will be needed on the next disk operation. To take advantage of this fact, the BIOS can keep information with its physical sector buffer as to which disk, track, and physical sector (if any) is represented in the buffer. Then, when reading, the BIOS need only perform physical disk reads when the data needed is not in the buffer.

When performing disk writes, the BIOS still needs to preread the physical sector for the same reasons discussed in Section 5.3.1. Once the physical sector is in the buffer, subsequent writes into that physical sector do not require additional prereads. To save additional disk accesses, do not write the sector to the disk until absolutely necessary. Section 5.3.4 discusses the conditions under which the physical sector must be written.

5.3.3 Track Buffering

Track buffering is a special case of disk buffering where the I/O is done a full track at a time. This method is quite good when sufficient memory for several full track buffers is available. This method employs the following differences from that discussed in Section 5.3.2. First, transferring an entire track is much more efficient than transferring a single sector. The rotational latency is incurred only once for the entire track, whereas if the track is transferred one sector at a time, the rotational latency occurs once per sector. On a typical diskette with 26 sectors per track, rotating at 6 revolutions per second, the difference in rotational latency per track is about 2 seconds versus a twelfth of a second. Of course, in applications where the disk is accessed purely randomly, there is no advantage because there is a low probability that more than one sector will be used from a given track. Note that such applications are extremely rare.

5.3.4 Least Recently Used Buffer Replacement

With any method of disk buffering using more than one buffer, it is necessary to have an algorithm to manage the buffers. A buffer should be filled when there is a request for a disk sector that is not presently in memory.

Generally, it is desirable to defer writing a buffer until it becomes necessary. Thus, several transfers can be done to a buffer for the cost of only one disk access, or two accesses if the buffer must be preread. There are four reasons why buffers must be written back to disk:

1. When a BIOS Write operation with mode=1 (write to directory sector) has been invoked. It is very important to the integrity of the CP/M-8000 file system that directory information on the disk is kept up to date. Therefore, all directory writes should be performed immediately.
2. A BIOS Flush Buffers operation. This BIOS function forces all disk buffers to be written. After performing a Flush Buffers, it is safe to remove a disk from its drive.
3. A disk buffer is needed, but all buffers are full. Therefore a buffer must be emptied to make it available for reuse.
4. A Warm Boot occurs. This is similar to number 2 above.

Case three above is the only case in which the BIOS writer has any discretion as to which buffer should be written. The best strategy is to write out the buffer that has been Least Recently Used. The fact that the contents of a buffer have not been accessed for some time is a fairly good indication that it will not be needed again soon.

5.3.5 The New Block Flag

As explained in Section 5.3.2, the BDOS allocates disk space to files in blocks of BLS bytes. When such a block is first allocated to a file, the information previously in that block need not be preserved. To enable the BIOS to take advantage of this fact, the BDOS uses a special parameter when calling the BIOS Write Function. This special parameter is indicated when register R5 contains the value 2 on a BIOS Write call, then the write being done is to the first sector of a newly allocated disk block. Therefore, the BIOS need not preread any sector of that block. If the BIOS performs disk buffering in units of BLS bytes, it can mark any free buffer as corresponding to the disk address specified in this write. This is because the contents of the newly allocated block are unimportant. If the BIOS uses a buffer size other than BLS, then the algorithm for taking full advantage of this information is more complicated.

Proper use of this flag reduces disk delay. Consider the case where one file is read sequentially and copied to a newly created file. Without this flag, every physical write would require a preread. With the flag, no physical write requires a preread. Thus, the number of physical disk operations is reduced by one third.

End of Section 5

Section 6
Installing and Adapting the Distributed BIOS and CP/M-8000

6.1 Overview

Digital Research supplies CP/M-8000 in a form suitable for booting on an Olivetti M20 system. If you have an Olivetti M20, you can read Section 6.2, which tells how to load the distributed system. Similarly, you can buy or lease some other machine that already runs CP/M-8000.

If you do not have an Olivetti M20, you can use the .REL files supplied with your distribution disks to bring up your first CP/M-8000 system. Section 6.3 discusses this process.

6.2 Booting on an Olivetti M20

The CP/M-8000 disk set distributed by Digital Research includes disks to boot and run CP/M-8000 on the Olivetti M20. You can use the distribution system boot disk without modification if you have an Olivetti M20 system with the following configuration:

- 256K memory (minimum required by the Olivetti memory management scheme)
- at least two double sided 5 1/4" floppy drives, or one double sided 5 1/4" floppy drive and one 5 1/4" hard disk.

To load CP/M-8000 on a system with two floppy drives, do the following:

1. Place the disk in the first floppy drive.
2. Press the SYSTEM RESET button (on the right hand side of the machine).
3. Type "F". This will cause the system to boot from floppy drive A:.

To load CP/M-8000 on a system with one floppy and one hard disk drive, do the following:

1. Insert the Olivetti PCOS™ system disk into the floppy drive.
2. Press the SYSTEM RESET button to boot PCOS.
3. Type "vf 10:" and a carriage return to format the hard disk.
4. Insert the CP/M-8000 distribution disk into the floppy drive.
5. Press the SYSTEM RESET button, then type "F". CP/M-8000 will boot.
6. Type "ERA C:*.*" and a carriage return to clear the hard disk directory.
7. You may then use PIP to transfer files to the hard disk.

6.3 Bringing Up CP/M-8000 Using the CPMSYS.REL Files

The CP/M-8000 distribution disks contain a copy of the CP/M-8000 operating system in relocatable object code form, for use in bringing up CP/M-8000 on any Z8000 system. The relocatable CP/M-8000 system is in the CPMSYS.REL file. This file contains the CCP and BDOS, but no BIOS. Release notes and/or a file named README.DOC describe the exact characteristics of the CPMSYS.REL file distributed on your disks. To bring up CP/M-8000 using the CPMSYS.REL file, you need:

- a method to down-load absolute data into your target system
- a computer capable of reading the CP/M-8000 distribution disks, such as the Olivetti M20
- a C language BIOS written for your target computer. This BIOS may be developed from the C language BIOS supplied on the CP/M-8000 distribution disks. Typically you will need to modify all the BIOS modules, and to write a new BIOSIO.C module.

Given the above items, you can use the following procedure to bring a working version of CP/M-8000 to your target system:

1. Compile your BIOS on the Olivetti M20.
2. Link CPMSYS.REL and your new BIOS.REL files on the Olivetti M20. Section 2 describes this process.
3. Down-load your new CP/M system created in step 2 to the target computer.

Now that you have a working version of CP/M-8000, you can use the tools provided with the distribution system for further development.

End of Section 6

Section 7

Cold Boot Automatic Command Execution

7.1 Overview

The Cold Boot Automatic Command Execution feature of CP/M-8000 allows you to configure CP/M-8000 so that the CCP will automatically execute a predetermined command line on cold boot. This feature can be used to start up turn-key systems.

7.2 Setting up Cold Boot Automatic Command Execution

The CBACE feature uses two global symbols: _autost, and _usercmd. These are both defined in the CCP, which uses them on cold boot to determine whether this feature is enabled. If you want to have a CCP command automatically executed on cold boot, you should include code in your BIOS's cold boot routine (at the label "bios") to perform the following:

1. Set the byte at _autost to the value 01H.
2. The command line to be executed must be placed in memory beginning at the _usercmd location. The command must be terminated with a NULL (00H) byte, and may not exceed 128 bytes in length. All alphabetic characters in the command line should be upper-case.

Once you write a BIOS that performs these two operations, you can build it into a CPM.SYS file as described in Section 2. This system, when booted, will execute the command you have built into it.

End of Section 7

Section 8
The PUTBOOT Utility

8.1 PUTBOOT Operation

The PUTBOOT utility copies a bootstrap loader program from a file to the system tracks of a disk.

8.2 Invoking PUTBOOT

Invoke PUTBOOT with a command of the form:

```
PUTBOOT <filename> <drive>
```

where

- <filename> is the name of the file to be written to the system tracks;
- <drive> is the drive specifier for the drive to which <filename> is to be written (letter in the range A-P.)

PUTBOOT writes the specified file to the system tracks of the specified drive. Sector skewing is not used; the file is written to the system tracks in physical sector number order.

Because the system tracks for the Olivetti M20 must have some special PCOS information on them, PUTBOOT contains logic to add that information to the system file placed on the system tracks.

PUTBOOT issues messages indicating successful or unsuccessful execution of the copy operation. The messages indicating successful execution are

```
Bootstrap file is x bytes.
```

This indicates the size of the boot file.

```
Bootstrap has been written.
```

This indicates the operation is complete.

The messages indicating errors in the PUTBOOT execution are

```
putboot: Illegal drive code <drive>
```

This indicates an illegal drive code in the <drive> specifier on the command line.

```
putboot: Can't open bootstrap file <filename>
```

This indicates that PUTBOOT cannot open the file specified in <filename> on the command line.

```
Bootstrap too big.
```

This indicates the file specified on the command line is too big to be copied to the system tracks.

```
Usage: putboot <filename> <drivecode>
```

This indicates that the command line had an argument error.

PUTBOOT uses BDOS calls to read the bootstrap loader program stored in the file specified in <filename>. PUTBOOT uses BIOS calls to write the bootstrap program to the system tracks. It refers to the OFF and SPT parameters in the Disk Parameter Block to determine the size of system track space. The source and command files for PUTBOOT are supplied on the distribution disks for CP/M-8000.

End of Section 8

APPENDIX A

Contents of Distribution Disks

This appendix describes briefly the files on the diskettes that contain CP/M-8000 as distributed by Digital Research.

File	Contents
AR8K.Z8K	Executable version of the archiver/librarian.
ASZ8K.PD	Predefinition file for the assembler.
ASZ8K.Z8K	Executable version of the assembler.
XCON.Z8K	Executable version of the XCON utility. The XCON utility translates from UNIDOT object file format to XOUT object file format.
BIOS.REL	A relocatable code file containing the BIOS for the Olivetti M20.
LDRBIOS.REL	A relocatable code file containing the loader BIOS for the Olivetti M20
BIOS.SUB	A submit file which creates a relocatable BIOS.REL file.
BIOSBOOT.8KN	BIOS boot code.
BIOSDEFS.8KN	BIOS assembly definitions for BIOS modules.
BIOSIF.8KN	BIOS interface code.
BIOSIO.8KN	BIOS I/O routines.
BIOSMEM.8KN	BIOS memory management routines.
BIOSTRAP.8KN	BIOS trap routines.
BIOS.C	C language source of Bootstrap and normal BIOS for the Olivetti M20.

File	Contents
SYSCALL.8KN	Interface for system calls in BIOS for Z8001.
COPY.Z8K	An executable version of the COPY utility.
CPM.SYS	Executable CP/M-8000 operating system file for the Olivetti M20.
CPMSYS.REL	Relocatable version of CP/M-8000 containing the CCP and BDOS modules.
CPMSYS2.REL	Relocatable version of CP/M-8000 for the Z8002. Contains the CCP and BDOS modules.
CPMLDR.REL	Relocatable bootstrap loader for the M20. Contains only BDOS module.
CPMLDR.SYS	The bootstrap loader for the M20. A copy of this is written to the system tracks using PUTBOOT.
CPMSYS.SUB	A submit file to create CPM.SYS.
DDT.Z8K	An executable version of DDT, the interactive debugger.
DUMP.Z8K	An executable version of the DUMP utility.
ED.Z8K	An executable version of the ED utility.
FORMAT.Z8K	An executable version of the disk formatter utility for the Olivetti M20.
FPE.O	Object file for floating point processor emulator. Linked into normal BIOS.
FPEDEP.O	Object file for processor dependent floating point processor emulator code. Linked into normal BIOS.

File	Contents
LDBDOS.REL	Loader BDOS relocatable object file.
LD8K.Z8K	Executable version of linker/loader
LIBCPM.a	C language runtime library for the Z8002. Functions execute in non-segmented mode.
LIBCPMS.a	C language runtime library for the Z8001. Functions execute in segmented mode.
OPT.O OPT.C OPT1.O OPT1.C	Object and C language versions of C language library optimization facilities. The file OPTION.H contains commentary explaining these facilities.
	The following files prefixed ".H" are C language declarations to be used in the BIOS or other user programs via the C language "include" directive.
OPTION.H	Declarations to eliminate unused C runtime library functions.
CTYPE.H	Macro definitions for ASCII coded integers.
ERRNO.H	Declarations of error codes.
PORTAB.H	Declarations for BIOS portability.
SETJMP.H	Declarations for setjmp and longjmp functions.
SIGNAL.H	Declarations for the signal function.
STDIO.H	Declaration of C standard I/O functions
XOUT.H	Declarations of CP/M-8000 object format.
ASSERT.H	Declaration of the ASSERT macro.
MAKELDR.SUB	Submit file to create CPMLDR.SYS.
MKPUTBT.SUB	Submit file to create PUTBOOT.
NMZ8K.Z8K	Executable version of the symbol table dump utility.
PIP.Z8K	An executable version of the PIP utility.

File	Contents
PUTBOOT.C	C language source of the Olivetti PUTBOOT utility.
PUTBOOT.Z8K	Executable version of the Olivetti PUTBOOT utility.
TRK.O	Object code specific to the Olivetti M20. PUTBOOT uses this code.
README	An ASCII file containing information relevant to this shipment of CP/M-8000.
SIZEZ8K.Z8K	Executable version of SIZEZ8K utility.
STARTUP.O	Startup routine for use with C programs. STARTUP must be the first object file linked.
STARTUP.8KN	STARTUP.8KN is for the Z8002.
STARTUP.8KS	STARTUP.8KS is for the Z8001.
STAT.Z8K	An executable version of the STAT utility.
XDUMP.Z8K	Executable version of XDUMP utility. XDUMP is like DUMP, and prints additional header and symbol table information.
ZCC.Z8K ZCC1.Z8K ZCC2.Z8K ZCC3.Z8K	The C language compiler and its overlays.

End of Appendix A

APPENDIX B
Sample BIOS Written in C

The listings in this appendix are also found on your CP/M-8000 distribution disk.

The Olivetti BIOS consists of both C language and assembly code. The C language code is conditionally compiled to produce either a loader BIOS for use with CPMLDR.SYS or a normal BIOS for use with CPM.SYS. Listing B-1 is the C language BIOS. Listings B-2 and B-3, BIOSASM.8KN and LBIOSASM.8KN, assemble the seven remaining assembler modules to form either a normal BIOS or a loader BIOS, based on the value of the label "LOADER".

Listing B-1. C Language BIOS

```
/*=======================================================================*/
/*+---------------------------------------------------------------------+*/
/*|                                                                     |*/
/*|      CP/M-8000(tm) BIOS for the OLIVETTI M20 (Z8000)                |*/
/*|                                                                     |*/
/*|      Copyright 1984, Digital Research Inc.                          |*/
/*|                                                                     |*/
/*+---------------------------------------------------------------------+*/
/*=======================================================================*/

/*-------------------------*/
/* Compilation information */
/*-------------------------*/
/*----------------------------------------------------------------------------*/
/*To compile bios.c for cpmldr.sys the command is: zcc -c -M1 -dLOADER bios.c */
/*This conditionally compiles bios.c leaving unrequired code out of the object*/
/*file.                                                                       */
/*----------------------------------------------------------------------------*/
/* The normal bios compile command for cpm.sys is: zcc -c -M1 bios.c          */
/* This will provide the full functionallity of the bios in the object file   */
/*----------------------------------------------------------------------------*/
/* By compiling bios.c with the command : zcc -c -M1 -dTRANSFER bios.c        */
/* You are provided with a bios object that allows the two floppy drives to   */
/* have two different formats. This is left purely as an example for the      */
/* the benefit of porting to a different format and can be modified.          */
/-----------------------------------------------------------------------------*/
/* By compiling bios.c with the command: zcc -c -M1 -dsect26 bios.c           */
/* 8" floppy disk support is provided by conditional compilation.             */
/*----------------------------------------------------------------------------*/
```

Listing B-1. (continued)

```
/* #define BAUD 1   */  /* Decommenting this define will conditionally compile*/
                        /* code setting the tty port to 1200 baud listening   */
                        /* XOFF on that port.                                 */

/* #define DEBUG 1  */  /* By decommenting this define hard disk debugging */
                        /* is enabled. This provides drive, block, and      */
                        /* track information to be printed on the console. */

char copyright[] = "Copyright 1984  Digital Research Inc.";

/* HISTORY
**
**      820803  S. Savitzky (Zilog) -- derived from 68000 EXORMACS bios
**
**      830614  F. Zlotnick (Zilog) -- removed initialization of iobyte
**                      upon each warmboot.  Changed seldisk to test for
**                      overflow of the dphtab (to fix the "dir d:" bug).
**
**      830804  F. Zlotnick (Zilog) -- Added conditional compilation for
**                      loader BIOS, which only needs a few of the BIOS
**                      functions.  The loader DOES require the definition
**                      of a context structure, for transfer of control to
**                      the system proper.
**
**      830804  F. Zlotnick (Zilog) -- Changed Disk Parameter Blocks to
**                      reflect new bootstrap method.
**
**      830809  F. Zlotnick (Zilog) -- Added escape character to keyboard
**                      map, as ctrl'['.
**
**      831205  K. Greenberg (Zilog) -- Fixed disk parameter table for hard
**                      drive C to point to dpb3, not dpb2 (80 trk floppy).
**
**      831212  K. Greenberg (Zilog) -- Modified disk parameter tables for
**                      hard disk to look more like floppies (fewer sectors
**                      but more tracks). This will fix the sector deblocking
**                      part of the bios to be compatible with both. Also
**                      switched to 4K allocation blocks and 512 direntries.
*/


/************************************************************************/
/************************************************************************/
/*                                                                      */
/*      I/O Device Definitions                                          */
/*                                                                      */
/************************************************************************/
/************************************************************************/
```

Listing B-1. (continued)

```
/***************************************************************************/
/* Define Interrupt Controller constants                                   */
/***************************************************************************/

/* The interrupt controller is an Intel 8259 left-shifted one bit          */
/* to allow for the word-alligned interrupt vectors of the Z8000.          */

/* === Assume that this is set up in the PROM === */

/***************************************************************************/
/*      Define the two USART ports                                         */
/***************************************************************************/

/* The USARTs are Intel 8251's */

#define KBD      0xA1                /* Keyboard USART base          */
#define RS232    0xC1                /* RS-232 terminal              */

#define SERDATA 0                    /* data port offset             */
#define SERCTRL 2                    /* control port offset          */
#define SERSTAT 2                    /* status port offset           */

#define SERINIT 0x37                 /* init (3 times)               */
#define SERRES  0x40                 /* reset                        */
#define SERMODE 0xEE                 /* mode (2 stop, even parity    */
                                     /*      parity disable, 8 bits  */
                                     /*      divide by 16            */
                                     /* DUBIOUS.  15??               */
#define TTYON  0x37                  /* cmd (no hunt, no reset,      */
                                     /*      RTS=0, error reset,     */
                                     /*      no break, rcv enable,   */
                                     /*      DTR=0, xmt enable       */
#define SERRRDY 0x02                 /* RCV ready bit mask           */
#define SERXRDY 0x01                 /* XMT ready bit mask           */
#define XON      0x11                /* Control- Q                   */
#define XOFF     0x13                /* Control- S                   */
/***************************************************************************/
/*      Define the counter/timer ports                                     */
/***************************************************************************/

/* The counter-timer is an Intel 8253 */

#define CT_232  0x121                /* counter/timer 0 -- RS232 baud rate   */
#define CT_KBD  0x123                /* counter/timer 1 -- kbd baud rate     */
#define CT_RTC  0x125                /* counter/timer 2 -- NVI (rt clock)    */
#define CT_CTRL 0x127                /* counter/timer control port           */
```

Listing B-1. (continued)

```
#define CT0CTL  0x36           /* c/t 0 control byte                     */
#define CT1CTL  0x76           /* c/t 1 control byte                     */
#define CT2CTL  0xB4           /* c/t 2 control byte                     */

/* control byte is followed by LSB, then MSB of count to data register  */
/* baud rate table follows:                                             */

#ifndef LOADER                 /* NOT needed by the Loader Bios          */

int baudRates[10] = {
                        1538,   /*    50 */
                         699,   /*   110 */
                         256,   /*   300 */
                         128,   /*   600 */
                          64,   /*  1200 */
                          32,   /*  2400 */
                          16,   /*  4800 */
                           8,   /*  9600 */
                           4,   /* 19200 */
                           2    /* 38400 */
                    };

#endif                         /* End Conditional */

/************************************************************************/
/* Define Parallel Port constants                                       */
/************************************************************************/

/* The parallel (printer) port is an Intel 8255 */

#define PAR_A   0x81           /* port A data          */
#define PAR_B   0x83           /* port B data          */
#define PAR_C   0x85           /* port C data          */
#define PARCTRL 0x87           /* control port         */

#define PARBSY  0x02           /* bit one (busy bit) needs to be low */
#define PARFLT  0x10           /* bit five (fault bit) needs to be high */
```

Listing B-1. (continued)

```
/***********************************************************************/
/***********************************************************************/
/*                                                                     */
/*              PROM AND HARDWARE INTERFACE                            */
/*                                                                     */
/***********************************************************************/
/***********************************************************************/

/***********************************************************************/
/* Define PROM I/O Addresses and Related Constants                     */
/***********************************************************************/
/*              SEE BIOSIO.8KN FOR THESE EXTERNALS                     */

extern int disk_io();   /* (char drive, cmd      -- disk I/O           */
                        /*  int  blk_count,                            */
                        /*  int  blk_num,                              */
                        /*  char *dest) -> int error?                  */

extern crt_put();       /* (char character)      -- put byte to CRT    */

extern cold_boot();     /* boot operating system                       */

#define DSKREAD  0      /* disk read command    */
#define DSKWRITE 1      /* disk write command   */
#define DSKFMT   2      /* disk format command  */
#define DSKVFY   3      /* disk verify command  */
#define DSKINIT  4      /* disk init. command   */


/***********************************************************************/
/* Define external I/O routines and addresses                          */
/***********************************************************************/
/*              SEE BIOSIF.8KN FOR THESE EXTERNALS                     */

extern output();                /* (port, data: int)    -- output      */
extern int input();             /* (port: int)          -- input       */

/***********************************************************************/
/* Define external memory management routines                          */
/***********************************************************************/
/*              SEE SYSCALL.8KN FOR THESE EXTERNALS                    */

extern mem_cpy();               /* (src, dest, len: long)-- copy data  */
extern long map_adr();          /* paddr = (laddr: long; space: int)   */

#define CDATA 0                 /* caller data space    */
#define CCODE 1                 /* caller code space    */
#define SDATA 2                 /* system data space    */
#define SCODE 3                 /* system code space    */
#define NDATA 4                 /* normal data space    */
#define NCODE 5                 /* normal code space    */
```

Listing B-1. (continued)

```
/************************************************************************/
/*      System Entry and Stack Pointer                                  */
/************************************************************************/

#define SYSENTRY        0x0b000006L     /* entry point */
#define SYSSTKPTR       0x0b00bffeL     /* system's stack pointer start */

/************************************************************************/
/*      Memory Region Table                                             */
/************************************************************************/

#ifndef LOADER                  /* NOT needed for the Loader Bios   */

struct mrt {    int  count;
                struct {long tpalow;
                        long tpalen;
                       }        regions[4];
           }
        memtab = {4,
                  0x0A000000L, 0x10000L,        /* merged    I and D */
                  0x08000000L, 0x10000L,        /* separated I       */
                  0x08000000L, 0x10000L,        /*             and D */
                  0x08000000L, 0x10000L         /* accessing I as  D */
                 };
#endif                          /* End conditional */
#ifdef LOADER                   /* NEEDED for the Loader Bios */

struct mrt {    int  count;
                struct {long tpalow;
                        long tpalen;
                       }        regions[1];
           }
        memtab = {1,
                  0x0B000000L, 0x0C000L,    /* system space: merged I and D */
                 };

struct context                  /* Startup context for user's program  */
        {
                short   regs[14];
                long    segstkptr;
                short   ignore;
                short   FCW;
                long    PC;
```

Listing B-1. (continued)

```
struct context context =

        {                               /* Regs 0-13 cleared, sp set up below   */
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                SYSSTKPTR,              /* Loaded system's stack pointer        */
                0,                      /* Ignore: value is zero                */
                0xD800,                 /* FCW: segmented system, VI, NVI set   */
                SYSENTRY                /* Entry point to system                */
        };
#endif                                  /* End conditional */

/****************************************************************************/
/* Set Exception Vector entry                                               */
/****************************************************************************/

extern long trapvec[];                  /* trap vector */

long setxvect(vnum, vval)
int vnum;
long vval;
{
          register long  oldval;

        oldval = trapvec[vnum];
        trapvec[vnum] = vval;

        return(oldval);

}
```

Listing B-1. (continued)

```
/*****************************************************************/
/*****************************************************************/
/*                                                               */
/*                       CHARACTER I/O                           */
/*                                                               */
/*****************************************************************/
/*****************************************************************/

/*****************************************************************/
/*      Generic Serial Port I/O Procedures                       */
/*****************************************************************/


/* define as extern the dirty flag, which is actually defined later     */
/* on in this file.  Used to flush the buffer at an opportune moment.   */

extern  int     tbdirty;

serinit(port)
int port;
{
        output(port+SERCTRL, SERINIT);
        output(port+SERCTRL, SERINIT);
        output(port+SERCTRL, SERINIT);

        output(port+SERCTRL, SERRES);
        output(port+SERCTRL, SERMODE);
        output(port+SERCTRL, TTYON);
#if BAUD                                    /* Conditional for 1200 baud    */
        output(CT_CTRL,CTOCTL);             /* Set baud rate genrator       */
        output(CT_232,baudRates[4]);        /* Modify for different speeds  */
        output(CT_232,0);                   /* Set for 1200 baud            */
#else
        /* === assume the PROM sets it up. === */
#endif                                      /* End conditional */
}


int serirdy(port)
int port;
{
        return(((input(port+SERSTAT) & SERRRDY) == SERRRDY) ? 0xFF : 0);
}
```

Listing B-1. (continued)

```
char serin(port)
int port;
{
        while (serirdy(port) == 0) ;
        return input(port+SERDATA);
}


int serordy(port)
int port;
{
        return(((input(port+SERSTAT) & SERXRDY) == SERXRDY) ? 0xFF : 0);
}


serout(port, ch)
int port;
char ch;
{
#if BAUD                             /* Conditional for 1200 baud and XOFF */
        while ( ((input(port + SERSTAT) & SERXRDY)
        != SERXRDY) | ((((input(port + SERDATA))
        & 0x7F) ^ XOFF) == 0));
        output(port + SERDATA, ch);
#else
        while ( (input(port + SERSTAT) & SERXRDY) != SERXRDY) ;
        output(port+SERDATA, ch);

#endif                               /* End conditional */
}


parordy(port)
int port;
{
     int status;
         status = (input(port));
         return (((status & PARBSY) != PARBSY) &&
                 ((status & PARFLT) == PARFLT) ? 0xFF : 0);
}

parout(port, ch)
int port;
char ch;
{
```

Listing B-1. (continued)

```
    register int i, status;

        i = 0;
        do
         {
          if (--i == 0)                                          /* only check for    */
                {printstr ("nrPrinter Timeout.nr");   /* printer ready a   */
                 return;                                          /* finite number of  */
                }                                                 /* times             */
         }
        while (!parordy(PAR_B));                                 /* if printer ready  */
        output (port, ch);                                       /* print character   */
        output (PARCTRL, 0x0A);                                  /* set strobe low    */
        output (PARCTRL, 0x0B);                                  /* set strobe high   */
}

/**************************************************************************/
/*      Olivetti keyboard translation table.                              */
/**************************************************************************/

#ifndef LOADER                          /* NOT needed for the Loader Bios        */

char kbtran[256] = {

/* Raw key codes for main keypad:

 SE       A    B    C    D    E    F    G    H    I    J    K    L    M    N
 O   P    Q    R    S    T    U    V    W    X    Y    Z    0    1    2    3
 4   5    6    7    8    9    -    ^    @    [    ;    :    ]    ,    .    /
*

/* main keyboard UNSHIFTED. */

0xDD,'\\','a','b','c','d','e','f','g','h','i','j','k','l','m','n',
 'o','p','q','r','s','t','u','v','w','x','y','z','0','1','2','3',
 '4','5','6','7','8','9','-','^','@','[',';',':',']',',','.','/',

/* main keyboard SHIFTED */

0xDE,'|','A','B','C','D','E','F','G','H','I','J','K','L','M','N',
 'O','P','Q','R','S','T','U','V','W','X','Y','Z',' ','!','"','#',
 '$','%','&','\'','(',')','=','~','`','{','+','*','}','<','>','?',

/* main keyboard CONTROL -- CTL B and C differ from Olivetti. */

0xA0,0x7F,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,
0x0F,0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0xE0,0xE1,0xE2,0xE3,
0xE4,0xE5,0xD6,0xE7,0xE8,0xE9,0xEA,0xEB,0x00,0x1B,0x1E,0x1F,0x1D,0xFE,0xFF,0xA4,
```

Listing B-1. (continued)

```
/* main keyboard COMMAND */

0xDF,0xF8,0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,
0x8E,0x8F,0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0xEC,0xED,0xEE,0xEF,
0xF0,0xF1,0xF2,0xF3,0xF4,0xF5,0xF6,0xF7,0x13,0x1C,0xFC,0xFD,0x9F,0xF9,0xFA,0xA5,

/* other keys

                SP     CR     S1     S2
   KEYPAD        .      0     00      1
                 2      3      4      5
                 6      7      8      9
                 +      -      *      /
*/

/* other keys UNSHIFTED  -- CR differs from Olivetti */

              ' ','r',0x7f,0x08,
              '.', '0',0xA6, '1',
              '2', '3', '4', '5',
              '6', '7', '8', '9',
              '+', '-', '*', '/',

/* other keys SHIFTED  -- CR differs from Olivetti */

              ' ','r',0xA8,0xA9,
              '.', '0',0xA6,0x1C,
             0x9A,0x1D,0x9B,0x9C,
             0x9D,0x1E,0x9E,0x1F,
             0x2B,0x2D,0x2A,0x2F,

/* other keys CONTROL */

              ' ','r',0xA8,0xA9,
             0x80,0x81,0x82,0x83,
             0x84,0x85,0x86,0x1B,
             0x88,0x89,0x8A,0x8B,
             0x8C,0x8D,0x8E,0x8F,


/* special -- substitute r for Olivetti's 0xAF. */

             'r','r','r','r'
};

#endif                              /* End conditional */
```

Listing B-1. (continued)

```
/*******************************************************************/
/*      specific I/O procedures for use with iobyte               */
/*******************************************************************/

/* CRT status, read, write routines */

int crtrs()
{
        return( serirdy(KBD));
}

#ifndef LOADER                  /* NOT needed for the Loader Bios */

char crtrd()
{
        return( kbtran[serin(KBD) & 0xff]);
}
#endif                          /* End conditional */

#ifdef LOADER                   /* Conditional for Loader Bios disable KBD */
#define crtrd nulrd
#endif                          /* End conditional */

int crtws()
{
        return(0xFF);
}

#define crtwr crt_put           /* output routine in PROM */

/* TTY status, read, write routines */

int ttyrs()
{
        return(serirdy(RS232));
}

char ttyrd()
{
        return(serin(RS232));
}

int ttyws()
{
        return(serordy(RS232));
}
```

Listing B-1. (continued)

```
ttywr(ch)
char ch;
{
        serout(RS232, ch);
}

/* LPT status, output routines */

int lptws()
{
        return (parordy (PAR_B));
}

lptwr(ch)       /* ARGSUSED */
char ch;
{
        parout (PAR_A, ch);
}

/*************************************************************************/
/*      generic device names, batch, and null devices                    */
/*************************************************************************/

/* the device names are the offset of the proper field in iobyte */

#define CON     0
#define READER  2
#define PUNCH   4
#define LIST    6


/* BATCH status, read, write routines */

#ifndef LOADER                  /* NOT needed by the Loader Bios        */
int batrs()
{
        int genstat();

        return genstat(READER);
}

char batrd()
{
        int genread();

        return genread(READER);
}
```

Listing B-1. (continued)

```
batwr(ch)
char ch;
{
        genwrite(LIST, ch);
}

#endif                          /* End Conditional */
#ifdef LOADER                   /* NEEDED for the Loader Bios          */
#define batrd nulrd
#define batrs nulst
#define batwr nulwr
#endif                          /* End conditional */

/* NULL status, read, write routines */

int nulst()
{
        return 0xFF;
}

char nulrd()
{
        return 0xFF;

nulwr(ch)                 /* ARGSUSED */
char ch;


/**********************************************************************/
/*      Generic I/O routines using iobyte                             */
/**********************************************************************/

/*
** IObyte itself.
*/

char iobyte = 0x41;;

/*
** Device operation tables.  DEVINDEX is the index into the
** table appropriate to a device (row) and its iobyte index (column)
**
**      nonexistent devices are mapped into NUL.
*/

#define DEVINDEX (((iobyte>>dev) & 3) + (dev * 2) )
```

Listing B-1. (continued)

```
int (*sttbl[16])() = {
        ttyrs, crtrs, batrs, nulst,     /* con    */
        ttyrs, nulst, nulst, nulst,     /* reader */
        ttyws, nulst, nulst, nulst,     /* punch  */
        ttyws, crtws, lptws, nulst      /* list   */
};

char (*rdtbl[16])() = {
        ttyrd, crtrd, batrd, nulrd,
        ttyrd, nulrd, nulrd, nulrd,
        nulrd, nulrd, nulrd, nulrd,
        nulrd, nulrd, nulrd, nulrd
};

int (*wrtbl[16])() = {
        ttywr, crtwr, batwr, nulwr,
        nulwr, nulwr, nulwr, nulwr,
        ttywr, nulwr, nulwr, nulwr,
        ttywr, crtwr, lptwr, nulwr
};

/*
** the generic service routines themselves
*/

int genstat(dev)
int dev;
{
        return( (*sttbl[DEVINDEX])() );
}

int genread(dev)
int dev;
{
        return( (*rdtbl[DEVINDEX])() );
}

genwrite(dev, ch)
int dev;
char ch;
{
        (*wrtbl[DEVINDEX])(ch);
}

#ifndef LOADER                  /* NOT needed for Loader Bios           */
```

Listing B-1. (continued)

```
/****************************************************************************/
/*      Error procedure for BIOS                                            */
/****************************************************************************/

bioserr(errmsg)
register char *errmsg;
{
        printstr("nrBIOS ERROR -- ");
        printstr(errmsg);
        printstr(".nr");
        while(1);
}

printstr(s)     /* used by bioserr */
register char *s;
{
        while (*s) {crtwr(*s); s += 1; };
}

#endif                       /* End conditional */

#ifdef DEBUG                 /* Conditional for Disk Debugging Hex output */
puthexd(i)      /* put a hex digit to crt */
int i;
{
        i &= 0xf;
        if (i < 10)
                crtwr(i + '0');
        else
                crtwr(i + 'a' - 10);
}

puthexv(i)      /* put an int in hex */
int i;
{
        puthexd(i >> 12);
        puthexd(i >> 8);
        puthexd(i >> 4);
        puthexd(i);
}
#endif                       /* End conditional */
```

Listing B-1. (continued)

```
/***************************************************************************/
/***************************************************************************/
/*                                                                         */
/*                           DISK I/O                                      */
/*                                                                         */
/***************************************************************************/
/***************************************************************************/


/***************************************************************************/
/* BIOS  Table Definitions                                                 */
/***************************************************************************/

struct dpb
{
        int     spt;             /* sectors per track                      */
        char    bsh;             /* block shift = log2(blocksize/128)      */
        char    blm;             /* block mask  = 2**bsh - 1               */
        char    exm;             /* extent mask                            */
        char    dpbjunk;         /*      dummy field to allign words       */
        int     dsm;             /* size of disk less offset, in blocks    */
        int     drm;             /* size of directory - 1                  */
        char    al0;             /* reservation bits for directory         */
        char    al1;             /* ...                                    */
        int     cks;             /* size of checksum vector = (drm+1)/4    */
        int     off;             /* track offset for OS boot               */
        char    psh;             /* log2(sectorsize/128)                   */
        char    psm;             /* physical size mask = 2**psh - 1        */
};


struct dph
{
        char    *xltp;           /* -> sector translation table            */
        int      dphscr[3];      /* scratchpad for BDOS                    */
        char    *dirbufp;        /* -> directory buffer (128 bytes)        */
struct  dpb     *dpbp;           /* -> disk parameter block                */
        char    *csvp;           /* -> software check vector (cks bytes)   */
        char    *alvp;           /* -> alloc vector ((dsm/8)+1 bytes)      */
};
```

Listing B-1. (continued)

```
/****************************************************************/
/*      Disk Parameter Blocks                                   */
/****************************************************************/

/*
** CP/M assumes that disks are made of 128-byte logical sectors.
**
** The Olivetti uses 256-byte sectors on its disks.  This BIOS buffers
** a track at a time, so sector address translation is not needed.
**
** Sample tables are included for several different disk sizes.
*/

/* === Olivetti has 3 floppy formats & a hard disk === */

#define SECSZ 128         /* CP/M logical sector size                 */
#define TRKSZ  32         /* track size for floppies, 1/2 track sz for hd */
#define PSECSZ 256        /* Olivetti physical sector size            */
#define PTRKSZ 16         /* physical track size                      */
#ifndef TRANSFER          /* Conditional for Normal bios              */
#define MAXDSK  3         /* max. number of disks                     */
#endif                    /* End conditional */
#ifdef  TRANSFER          /* Tranfer Conditional needs an extra dpb define*/
#define MAXDSK  4         /* Disk 4 is a pseudonym for disk 2, with   */
#endif                    /*   an old-style dpb to rescue those files.  */
                          /* End conditional */

/*****   spt, bsh, blm, exm, jnk,  dsm, drm,  al0, al1, cks, off, psh, psm */

struct dpb dpb0=          /* --- 1 side, 16*256 sector, 35 track.  140kb --- */
       { 32,  4,  15,  1,  0,   64,  63, 0xC0,  0,  16,   3};
struct dpb dpb1=          /* --- 2 side, 16*256 sector, 35 track.  280kb --- */
       { 32,  4,  15,  1,  0,  134,  63, 0xC0,  0,  16,   3};
struct dpb dpb2=          /* --- 2 side, 16*256 sector, 80 track.  640kb --- */
       { 32,  4,  15,  0,  0,  314,  63, 0xC0,  0,  16,   3};
struct dpb dpb3=          /* --- 6 side, 32*256 sector, 180 trk.  8640kb --- */
       { 32,  5,  31,  1,  0, 2154, 511, 0xf0,  0,  0,    3};
#ifdef TRANSFER           /* Conditional Tranfer dpb defined here        */
struct dpb dpb4=          /* --- 2 side, 16*256 sector, 35 track.  280kb --- */
       { 32,  4,  15,  1,  0,  120,  63, 0xC0,  0,  16,  10};
#endif                    /* End conditional */
/*              bls = 2K      dsm = (disk size - 3 reserved tracks) / bls */
/*              bls = 4K for hard disk (8640 - 24)   4                    */
#ifdef SECT26             /* Conditional for 8" floppy drives     */

/* === The Olivetti does not have 26-sector disks, but many people do.
**     The following parameter blocks are provided for their use.
*/
```

Listing B-1. (continued)

```
struct dpb dpb8=        /* --- 1 side, 26*128 sector, 77 trk --- */
        { 26,   3,   7,    0,   0,   242,  63, 0xC0,   0,  16,   2};
struct dpb dpbD=        /* --- 1 side, 26*256 sector, 77 trk --- */
        { 52,   4,  15,    0,   0,   242,  63, 0xC0,   0,  16,   2};

#endif                  /* End conditional */

/*************************************************************************/
/*      BDOS Scratchpad Areas                                            */
/*************************************************************************/

char    dirbuf[SECSZ];


char    csv0[16];
char    csv1[16];
char    csv2[32];
#ifdef TRANSFER         /* For Transfer conditional */
char    csv3[16];
#endif                  /* End conditional */


char    alv0[32];       /* (dsm0 / 8) + 1       */
char    alv1[32];       /* (dsm1 / 8) + 1       */
char    alv2[2002];     /* (dsm2 / 8) + 1       */
#ifdef TRANSFER         /* For Transfer conditional */
char    alv3[32];
#endif                  /* End conditional */


/*************************************************************************/
/* Sector Translate Table                                                */
/*************************************************************************/


#ifdef SECT26           /* Conditional for 8" floppy drives */

/* === The Olivetti does not have 26-sector disks, but many people do.
**     The following translate table is provided for their use.
*/

char    xlt26[26] = {  1,  7, 13, 19, 25,  5, 11, 17, 23,  3,  9, 15, 21,
                       2,  8, 14, 20, 26,  6, 12, 18, 24,  4, 10, 16, 22 };

#endif                  /* End conditional */

char    xlt16[32] = { 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,
                     17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32};
```

Listing B-1. (continued)

```
/****************************************************************************/
/* Disk Parameter Headers                                                   */
/*                                                                          */
/* Three disks are defined: dsk a: diskno=0, drive 0                        */
/*                          dsk b: diskno=1, drive 1                        */
/*                          dsk c: diskno=2, drive 10                       */
/****************************************************************************/

#ifndef TRANSFER            /* Normal bios dph conditional */
struct dph dphtab[3] =
        { {xlt16, {0, 0, 0}, dirbuf, &dpb1, csv0, alv0},     /*dsk a*/
          {xlt16, {0, 0, 0}, dirbuf, &dpb1, csv1, alv1},     /*dsk b*/
          {xlt16, {0, 0, 0}, dirbuf, &dpb3, csv2, alv2},     /*dsk c*/
        };
#endif                      /* End conditional */

#ifdef TRANSFER             /* Trnafer conditional with extra dph */
struct dph dphtab[4] =
        { {xlt16, {0, 0, 0}, dirbuf, &dpb1, csv0, alv0},     /*dsk a*/
          {xlt16, {0, 0, 0}, dirbuf, &dpb1, csv1, alv1},     /*dsk b*/
          {xlt16, {0, 0, 0}, dirbuf, &dpb3, csv2, alv2},     /*dsk c*/
          {xlt16, {0, 0, 0}, dirbuf, &dpb4, csv3, alv3},     /*dsk d*/
        };
#endif                      /* End conditional */

/****************************************************************************/
/*      Currently Selected Disk Stuff                                       */
/****************************************************************************/

int settrk, setsec, setdsk;     /* track, sector, disk #                    */
long setdma;                    /* dma address with segment info: long      */


char    trkbuf[TRKSZ * SECSZ];  /* track buffer          */
int     tbvalid = 0;            /* track buffer valid    */
int     tbdirty = 0;            /* track buffer dirty    */
int     tbtrk;                  /* track buffer track #  */
int     tbdsk;                  /* track buffer disk #   */
int     dskerr;                 /* disk error            */
```

Listing B-1. (continued)

```
/****************************************************************************/
/*      Disk I/O Procedures                                                 */
/****************************************************************************/


dskxfer(dsk, trk, bufp, cmd)     /* transfer a disk track */
register int  dsk, trk, cmd;
register char *bufp;
/*
                This is a handy place to keep notes on Olivetti block
        numbering. For a floppy, bits 3-0 are sector, bit 4 is side,
        and high-order bits are track. We define a floppy to have
        twice as many sectors as there are on a track: thus, the
        sector number overflows to the side bit and all is well. On
        the hard disk, bits 4-0 are sector (there are 32 per track),
        and the high-order bits are (track*6)+surface, where surface
        is in the range 0..5. To make the indexing of trkbuf consistent,
        we define a hard disk to have only 32 logical (16 physical)
        sectors per track, like a floppy. Thus we will transfer only
        half a track to/from the buffer at a time, and the logical
        sector number will overflow into the real high-order bit of
        the sector number. This works because we will always move
        half a track at a time. The tracks and surfaces simply take
        care of themselves, incrementing through the surfaces and
        effectively minimizing seeks.
*/
{
        int blknum;

        if (dsk==2)
                dsk = 10;               /* convert hard disk drive # */
#ifdef TRANSFER                         /* Conditional reassignment for Transfer */
        if(dsk==3) dsk = 1;             /* for transfer disks    */
#endif                                  /* End conditional */
        dskerr=0;                       /* assume no error            */
                                        /* do transfer                */
#ifdef DEBUG                            /* Conditional DEBUG output   */
        blknum = trk*PTRKSZ;
        printstr("nxfer block ");
        puthexv(blknum);
        printstr(" unit ");
        puthexd(dsk);
        printstr(" track ");
        puthexv(trk);
        if (cmd == DSKREAD)
                printstr(" read");
```

Listing B-1. (continued)

```
        else
                printstr(" write");
        crtwr(10); crtwr(13);
#endif                                  /* End conditional */
        if (0 != disk_io(dsk, cmd, PTRKSZ, trk*PTRKSZ, map_adr((long)bufp,0)))
                dskerr=1;
        }
}


#define wrongtk ((! tbvalid) || (tbtrk != settrk) || (tbdsk != setdsk))
#define gettrk  if (wrongtk) filltb()

#ifndef LOADER                          /* NOT needed for Loader Bios */

flush()
{

        if ( tbdirty && tbvalid ) dskxfer(tbdsk, tbtrk, trkbuf, DSKWRITE);

        tbdirty = 0;



#endif                                  /* End conditional */


filltb()
{

#ifndef LOADER                          /* NOT needed by Loader Bios */
        if ( tbvalid && tbdirty ) flush();
#endif                                  /* End conditional */
        dskxfer(setdsk, settrk, trkbuf, DSKREAD);

        tbvalid = 1;
        tbdirty = 0;
        tbtrk   = settrk;
        tbdsk   = setdsk;



dskread()

         register char  *p;

        gettrk;
        p = &trkbuf[SECSZ * (setsec-1)];
```

Listing B-1. (continued)

```
        /* transfer between memory spaces.  setdma is physical address */

        mem_cpy(map_adr((long)p, CDATA), setdma, (long)SECSZ);

        return(dskerr);
}

#ifndef LOADER                  /* NOT needed by Loader Bios it doesn't write */

dskwrite(mode)
char mode;
{
         register char  *p;

        gettrk;
        p = &trkbuf[SECSZ * (setsec-1)];

        /* transfer between memory spaces.  setdma is physical address */

        mem_cpy(setdma, map_adr((long) p, CDATA), (long)SECSZ);
        tbdirty = 1;
        if ( mode == 1 ) flush();
        return(dskerr);
}

#endif                          /* End conditional */

char sectran(s, xp)
int      s;
char     *xp;
{
        if (xp != 0) return xp[s]; else return s;
}

struct dph *seldisk(dsk, logged)
register char dsk;
    char logged;
{
        register struct dph *dphp;

        if (dsk > MAXDSK) return(0L);
        setdsk = dsk;
        dphp = &dphtab[dsk];
        if (dphp >= dphtab + (sizeof(dphtab)/sizeof(struct dph)) ) return(0L);
        if ( ! logged )
        {

                /* === disk not logged in. select density, etc. === */

        }
        return(dphp);
```

Listing B-1. (continued)

```
/****************************************************************/
/****************************************************************/
/*                                                              */
/*                        BIOS PROPER                           */
/*                                                              */
/****************************************************************/
/****************************************************************/



biosinit()
{
#ifdef DEBUG            /* Conditional banner for DEBUG */
        printstr("rnCP/M-8000:  Olivetti M20 BIOS DEBUG");
#endif                  /* End conditional */
        /* serinit(KBD);*/      /* DON'T init keyboard serial port     */
        serinit(RS232);         /* init rs232 serial port              */

        tbvalid = 0;            /* init disk flags                     */
        tbdirty = 0;

                /* Following reset of iobyte on each warm boot has been */
                /* removed, so that STAT can reassign devices.  iobyte  */
                /* is now initialized on cold boot only.                */
        /* iobyte = 0x41; */    /* con, list = CRT; rdr, punch = TTY   */
}

/* In the LOADER bios, the main routine is called "bios", not "_bios" */
#ifdef LOADER                   /* Loader Bios conditional */
#define _bios   bios
#endif                          /* End conditional */

long _bios(d0, d1, d2)
int     d0;
long    d1, d2;
{

        switch(d0)
        {
                case 0:                                 /* INIT         */
                        biosinit();
                        break;

#ifndef LOADER          /* Normal Bios use */
                case 1:                                 /* WBOOT        */
                        wboot();
                        break;

#endif                  /* End conditional */
```

Listing B-1. (continued)

```
                case 2:                           /* CONST            */
                         return(genstat(CON));
                         break;

                case 3:                           /* CONIN            */
                         return(genread(CON));
                         break;

                case 4:                           /* CONOUT           */
                         genwrite(CON, (char)d1);
                         break;

#ifndef LOADER           /* Normal Bios use */

                case 5:                           /* LIST             */
                         genwrite(LIST, (char)d1);
                         break;

                case 6:                           /* PUNCH            */
                         genwrite(PUNCH, (char)d1);
                         break;

                case 7:                           /* READER           */
                         return(genread(READER));
                         break;

                case 8:                           /* HOME             */
                         settrk = 0;
                         break;

#endif                   /* End conditional */

                case 9:                           /* SELDSK           */
                         return((long)seldisk((char)d1, (char)d2));
                         break;

                case 10:                          /* SETTRK           */
                         settrk = (int)d1;
                         break;

                case 11:                          /* SETSEC           */
                         setsec = (int)d1;
                         break;

                case 12:                          /* SETDMA           */
                         setdma = d1;
                         break;

                case 13:                          /* READ             */
                         return(dskread());
                         break;
```

Listing B-1. (continued)

```
#ifndef LOADER          /* Normal Bios use */
                case 14:                         /* WRITE        */
                        return(dskwrite((char)d1));
                        break;

                case 15:                         /* LISTST       */
                        return(genstat(LIST));
                        break;

#endif                  /* End conditional */

                case 16:                         /* SECTRAN      */
                        return(sectran((int)d1, (char*)d2));
                        break;

                case 18:                         /* GMRTA        */
                        return((long)&memtab);
                        break;

#ifndef LOADER          /* Normal Bios use */
                case 19:                         /* GETIOB       */
                        return((long)iobyte);
                        break;

                case 20:                         /* SETIOB       */
                        iobyte = (char)d1;
                        break;

                case 21:                         /* FLUSH        */
                        flush();
                        return((long)dskerr);
                        break;

#endif                  /* End conditional */

                case 22:                         /* SETXVECT     */
                        return(setxvect((int)d1, d2));
                        break;

        } /* end switch */

        return(0);

} /* end bios procedure */

/* End of C Bios */
```

Listing B-2. Normal BIOS Assembler

```
;*********************************************
;Build the assembly modules using conditionals
;
__text: .sect
;
;by setting the value of the label LOADER false
;(0) the normal Bios code will be generated
;while setting the label to true (1) will
;provide the loader Bios code.
;
LOADER .equ     0       ;  1 or 0 which ever
;
;
        .input "biosdefs.8kn"
        .input "biosboot.8kn"
        .input "biosif.8kn"
        .input "biosio.8kn"
        .input "biosmem.8kn"
        .input "biostrap.8kn"
        .input "syscall.8kn"


;****************************************************
;*
;* Data
;*
;****************************************************

__bss:  .sect

_sysseg: .block 2       ;system segment
_usrseg: .block 2       ;user segment
_sysstk: .block 4       ;system stack pointer
_psap:   .block 4       ;program status area ptr
```

Listing B-3. (continued)

```
;*****************************************************
;*
;* Trap vector table
;*
;*      entries 0..31 are misc. system traps
;*      entries 32..47 are system calls 0..15
;*
;*****************************************************


_trapvec:
        .block NTRAPS*4

;*****************************************************
;*****************************************************
;       ***** 8/15/84 R.F.W. *****
```

Listing B-4. BIOS Assembly Language Definitions

```
;********* biosdefs.8kn cpm.sys +cpmldr.sys***********
;*
;*      Assembly language definitions for
;*      CP/M-8000 (tm) BIOS
;*
;* 821013 S. Savitzky (Zilog) -- created.
;*


;****************************************************
;*
;* System Calls and Trap Indexes
;*
;****************************************************

XFER_SC .equ    1
BIOS_SC .equ    3
BDOS_SC .equ    2
MEM_SC  .equ    1
DEBUG_SC .equ   0

;* the traps use numbers similar to those in the
;* 68K version of P-CP/M

NTRAPS  .equ    48      ;total number of traps
SC0TRAP .equ    32      ;trap # of system call 0

                    ;Z8000 traps
EPUTRAP .equ    1       ;EPU (floating pt. emulator)
SEGTRAP .equ    2       ;segmentation (68K bus err)
NMITRAP .equ    0       ;non-maskable int.
PITRAP  .equ    8       ;priviledge violation
                    ;Interrupts, etc.
TRACETR .equ    9       ; trace
```

Listing B-4. (continued)

```
;*************************************************************
;*
;* C Stack frame equates
;*
;*      A C stack frame consists of the PC on top,
;*      followed by the arguments, leftmost argument first.
;*
;*      The caller adjusts the stack on return.
;*      Returned value is in r7 (int) or rr6 (long)
;*
;*************************************************************

PCSIZE   .equ    2                   ;PC size non-segmented
INTSIZE  .equu   2                   ;INT data type size
LONGSIZE .equ    4                   ;LONG data type size

ARG1     .equ    PCSIZE              ;integer arguments
ARG2     .equ    ARG1+INTSIZE
ARG3     .equ    ARG2+INTSIZE
ARG4     .equ    ARG3+INTSIZE
ARG5     .equ    ARG4+INTSIZE


;*************************************************************
;*
;* Segmented Mode Operations
;*
;*      NOTE:   segmented indirect-register operations
;*              can be done by addressing the low half
;*              of the register pair.
;*
;*************************************************************

SEG      .MACRO             ; START segmented mode
                            ; r0 destroyed.

         ldctl   r0,FCW
         set     r0,#15
         ldctl   FCW,r0

         .ENDM


NONSEG   .MACRO             ; END segmented mode
                            ; r0 destroyed.

         ldctl   r0,FCW
         res     r0,#15
         ldctl   FCW,r0

         .ENDM
```

Listing B-4. (continued)

```
scall   .MACRO          ;(segaddr)      segmented CALL

        .word   05F00h
        .long   ?1

        .ENDM


sscall  .MACRO          ;(|segaddr|)    short segmented CALL

        .word   05F00h
        .word   ?1

        .ENDM


;*******************************************************
;*
;* System Call Trap Handler Stack Frame
;*
;*******************************************************

cr0     .equ    0               ;WORD   caller r0
cr1     .equ    cr0+2           ;WORD   caller r1
cr2     .equ    cr1+2           ;WORD   caller r2
cr3     .equ    cr2+2           ;WORD   caller r3
cr4     .equ    cr3+2           ;WORD   caller r4
cr5     .equ    cr4+2           ;WORD   caller r5
cr6     .equ    cr5+2           ;WORD   caller r6
cr7     .equ    cr6+2           ;WORD   caller r7
cr8     .equ    cr7+2           ;WORD   caller r8
cr9     .equ    cr8+2           ;WORD   caller r9
cr10    .equ    cr9+2           ;WORD   caller r10
cr11    .equ    cr10+2          ;WORD   caller r11
cr12    .equ    cr11+2          ;WORD   caller r12
cr13    .equ    cr12+2          ;WORD   caller r13
nr14    .equ    cr13+2          ;WORD   normal r14
nr15    .equ    nr14+2          ;WORD   normal r15
scinst  .equ    nr15+2          ;WORD   SC instruction
scfcw   .equ    scinst+2        ;WORD   caller FCW
scseg   .equ    scfcw+2         ;WORD   caller PC SEG
scpc    .equ    scseg+2         ;WORD   caller PC OFFSET
FRAMESZ .equ scpc+2
```

Listing B-5. Olivetti Bootstrap Initialization

```
;******** biosboot.8kn for cpm.sys + cpmldr.sys*****
;*      Copyright 1984, Digital Research Inc.
;*
;* 821013 S. Savitzky (Zilog) -- adapt for nonseg.
;* 820930 S. Savitzky (Zilog) -- created
;* 840813 R. Weiser (DRI) -- conditional assembly
;*
;****************************************************

__text: .sect

;****************************************************
;*
;* NOTE -- THIS CODE IS HIGHLY SYSTEM-DEPENDENT
;*
;*      This module contains both the bootstrap
;*      writer, and the code that receives control
;*      after being booted.
;*
;*      The main function of the latter is to make
;*      sure that the system, whose entry point is
;*      called "bios", is passed a valid stack
;*      and PSA pointer.
;*
;*      Although this code runs segmented, it must
;*      be linked with non-segmented code, so it
;*      looks rather odd.
;*
;****************************************************
```

Listing B-5. (continued)

```
;************************************************************
;*
;*      CP/M - 8000 on the Olivetti M20.
;*
;*      Olivetti's peculiar format, has a lot of
;*      Olivetti's file system in it.
;*
;*      Track 0 is unused except for sector 0, since
;*      it is single density and thus has smaller
;*      sectors.
;*
;*      A total of 10 tracks are reserved from CP/M,
;*      leaving 9 tracks for the system proper.
;*
;*      The first sector on track 1 is the PCOS file
;*      descriptor block; the second is the boot file
;*      header and the start of the system code.
;*
;*      This leaves something under 28K for the
;*      system (BIOS+BDOS+CCP).  It is assumed that
;*      the system starts at its lowest address,
;*      and that data follows immediately after code.
;*
;*      For now, we assume that the system starts at
;*              <<11>>0000 (hex) for normal system
;*              <<10>>0000 (hex) for boot system
;*
;************************************************************


;************************************************************
;*
;*   Globals
;*
;************************************************************

        .if LOADER
        .global _startld  ;entry to read system tracks
        .endif
```

Listing B-5. (continued)

```
;*************************************************************
;*
;* Externals
;*
;*************************************************************

        .global bios
        .if LOADER
        .else           ; no warm boots in loader bios
        .global _wboot
        .endif

;*************************************************************
;*
;* Constants
;*
;*************************************************************

        .if LOADER
BOOTSYS .equ    0A000000h       ;system address on boot
BOOTSTK .equ    BOOTSYS+0BFFEh  ;system stack top on boot
        .else
SYSTEM  .equ    0B000000h       ; system address
SYSSTK  .equ    SYSTEM+0BFFEh   ; system stack top
        .endif

BPT     .equ    16              ; #blocks in a track
BPS     .equ    256             ; #bytes in a sector
NBLKS   .equ    9*16            ; #blocks in boot
HDRSIZE .equ    24              ; #bytes in header
FILSIZE .equ    256*(NBLKS-1)   ; file data size
SYSSIZE .equ    FILSIZE-HDRSIZE ; total system size
S1SIZE  .equ    BPS-HDRSIZE     ; data in sector 1

SEG4    .equ    04000000h
SEG2    .equ    02000000h

SYSPSA  .equ    SEG2+100h       ; system PSA
BOOTPSA .equ    SEG4+100h       ; PSA in PROM for boot

sscall  .macro          ;short segmented call
        .word   05f00h
        .word   ?1
        .endm
```

Listing B-5. (continued)

```
;*********************************************************
;*
;* Entry Points and post-boot Initialization
;*
;*********************************************************


;* transfer vector

        .if LOADER
        .else           ; no warm boot in the loader bios
        jr      wboot
        .endif
        jr      entry

;* post-boot init.

entry:                          ;SEGMENTED
        .if LOADER
_startld:
        .endif


        DI      VI,NVI

        .if LOADER
        ldl     rr14, #BOOTSTK  ;init boot stack pointer
        .else
        ldl     rr14, #SYSSTK   ;init normat stack pointer
        .endif
        ldl     rr2, #SYSPSA    ; copy PROM's PSA
        ldctl   r4, psapseg
        ldctl   r5, psapoff
        ld      r0, #570/2
        ldir    @r2, @r4, r0

        ldl     rr2, #SYSPSA    ; shift PSA pointer
        ldctl   psapseg, r2
        ldctl   psapoff, r3
```

Listing B-5. (continued)

```
        ld      r2,#142h        ;CROCK-- turn off
        ld      r3,#1feh        ; usart interrupts
        out     @r2,r3

        ldar    r2, $           ; go
        ld      r3,#bios
        jp      @r2


        .if LOADER
        .else                   ;no warmboot in loader bios

wboot:  ldar    r2, $
        ld      r3,#_wboot
        jp      @r2
        .endif
```

Listing B-6. BIOS Assembly Language Interface

```
;********* biosif.8kn for cpm.sys + cpmldr.sys *******
;*      Copyright 1984, Digital Research Inc.
;*
;* Assembly language interface for CP/M-8000(tm) BIOS
;*        ----- System-Independent -----
;*
;* 821013 S. Savitzky (Zilog) -- split into modules
;* 820913 S. Savitzky (Zilog) -- created.
;* 840811 R. Weiser (DRI)    -- conditional assembly
;*

__text: .sect

;*****************************************************
;*
;* NOTE
;*      The C portion of the BIOS is non-segmented.
;*
;*      This assembly-language module is assembled
;*      non-segmented, and serves as the interface.
;*
;*      Segmented operations are well-isolated, and
;*      are either the same as their non-segmented
;*      counterparts, or constructed using macros.
;*      The resulting code looks a little odd.
;*
;*****************************************************
```

Listing B-6. (continued)

```
;****************************************************
;*
;* Externals
;*
;****************************************************

        .global _biosinit       ;C portion init

        .if  LOADER             ; If LOADER is True then
        .global _ldcpm          ; Load the system into memory

        .else                   ; else its the normal bios

        .global _flush          ;Flush buffers

        .global ccp             ;Command Processor

        .endif                  ; end conditional

        .global _trapinit       ;trap startup

        .global _psap, _sysseg, _sysstk

;****************************************************
;*
;* Global declarations
;*
;****************************************************

        .global bios            ; initialization
        .if LOADER              ; If Loader stub out _wboot
        .else
        .global _wboot          ; warm boot
        .endif
        .global _input          ; input a byte
        .global _output         ; output a byte
```

Listing B-6. (continued)

```
;*****************************************************************
;*
;* Bios Initialization and Entry Point
;*
;*      This is where control comes after boot.
;*      If (the label LOADER is true 1)
;*      Control is transferred to -ldcpm
;*      else
;*      Control is transferred to the ccp.
;*
;*      We get here from bootstrap with:
;*              segmented mode
;*              valid stack pointer
;*              valid PSA in RAM
;*
;*****************************************************************

bios:

; enter in segmented mode.
; Get system (PC) segment into r4

        DI      VI,NVI
        calr    kludge  ; get PC segment on stack
kludge: popl    rr4,@r14

; get PSAP into rr2.

        ldctl   r2, PSAPSEG
        ldctl   r3, PSAPOFF

; go non-segmented.  save PSAP, system segment,
; system stack pointer (in system segment, please)

        NONSEG

        ldl     _psap, rr2
        ld      _sysseg, r4
        ld      r14,_sysseg
        ldl     _sysstk, rr14

        .if LOADER
        .else
;  set up system stack so that a return will warm boot
```

Listing B-6. (continued)

```
        push    @r15,#_wboot
        .endif
; set up traps, then enable interrupts

        call    _trapinit
        EI      VI,NVI

; set up C part of Bios

        call    _biosinit

; Turn control over to command processor
        .if LOADER
        jp      _ldcpm  ; do Program load
        .else
        jp      ccp

;*******************************************************
;*
;* Warm Boot
;*
;*      flush buffers and initialize Bios
;*      then transfer to CCP
;*
;*******************************************************

_wboot:
        call    _flush
        call    _biosinit
        ldl     rr14,_sysstk
        jp      ccp

        .endif
```

Listing B-6. (continued)

```
;*******************************************************
;*
;* I/O port operations
;*
;* int = input(port: int)
;* output (port, data: int)
;*
;*******************************************************

_input:
        ld      r2,ARG1(r15)
        subl    rr6,rr6
        inb     rl7,@r2
        ldb     rl6,rl7
        ret


_output:
        ld      r2,ARG1(r15)
        ld      r3,ARG2(r15)
        outb    @r2,rl3
        ret

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

Listing B-7. BIOS I/O Routines

```
;******** biosio.8kn for cpm.sys + cpmldr.sys *******
;*       Copyright 1984, Digital Research Inc.
;*
;*       I/O routines for CP/M-8000(tm) BIOS
;*       for Olivetti M20 (Z8001) system.
;*
;* 821013 S. Savitzky (Zilog) -- created.
;* 840815 R. Weiser (DRI) -- conditional assembly
;*

__text: .sect

;****************************************************
;*
;* NOTE The Olivetti PROM routines are segmented.
;*       The C portion of the BIOS is non-segmented.
;*
;*       This assembly-language module is assembled
;*       non-segmented, and serves as the interface.
;*
;*       Segmented operations are well-isolated, and
;*       are either the same as their non-segmented
;*       counterparts, or constructed using macros.
;*
;****************************************************


;****************************************************
;*
;* Global declarations
;*
;****************************************************


        .global _disk_io
        .global _crt_put
        .global _cold_boot
```

Listing B-7. (continued)

```
;*******************************************************
;*
;* Prom Subroutine Access
;*
;*******************************************************

_disk_io:       ;err=disk_io(drv, cmd, count, blk, addr)

        dec     r15,#14         ;save registers
        ldm     @r15,r8,#7

        ldb     rh7,14+ARG1+1(r15)      ;get args
        ldb     rl7,14+ARG2+1(r15)
        ld      r8, 14+ARG3(r15)
        ld      r9, 14+ARG4(r15)
        ldl     rr10,14+ARG5(r15)

                        ;rh7 = drive #
                        ;rl7 = command
                        ;r8  = block count
                        ;r9  = block number
                        ;rr10 = segmented address

        SEG
        scall   84000068h
        NONSEG
                        ;r8  = block count not transferred
                        ;rh7 = #retries
                        ;rl7 = final error code (RETURNED)
                        ;rh6 = error retried

        and     r7,#0FFh        ;value returned in r7

        ldm     r8,@r15,#7      ;restore regs
        inc     r15,#14
        ret


_crt_put:       ;crt_put(char)

        dec     r15,#14         ;save registers
        ldm     @r15,r8,#7

        ld      r1,14+ARG1(r15) ;get arg in r1

        SEG                     ; SEG clobbers r0
        ld      r0,r1           ;rl0 = char
        scall   84000080h
        NONSEG
```

Listing B-7. (continued)

```
        ldm     r8,@r15,#7      ;restore regs
        inc     r15,#14
        ret

_cold_boot:

        SEG
        scall   8400008Ch
        NONSEG
        ret
```

Listing B-8. Memory Management BIOS

```
;******* biosmem.8kn for cpm.sys + cpmldr.sys ********
;*      Copyright 1984, Digital Research Inc.
;*
;*      Memory Management for CP/M-8000(tm) BIOS
;*      for Olivetti M20 (Z8001) system.
;*
;* 821013 S. Savitzky (Zilog) -- split modules
;* 820913 S. Savitzky (Zilog) -- created.
;* 840815 R. Weiser (DRI) -- conditional assembly
;*

__text: .sect

;*****************************************************
;*
;*      This module copies data from one memory space
;*      to another.  The machine-dependent parts of
;*      the mapping are well isolated.
;*
;*      Segmented operations are well-isolated, and
;*      are either the same as their non-segmented
;*      counterparts, or constructed using macros.
;*
;*****************************************************


;*****************************************************
;*
;* Global declarations
;*
;*****************************************************

        .global _sysseg, _usrseg, _sysstk, _psap

        .global memsc

;*****************************************************
;*
;* Externals
;*
;*****************************************************

        .global xfersc
```

Listing B-8. (continued)

```
;************************************************************
;*
;*  System/User Memory Access
;*
;*  _mem_cpy( source, dest, length)
;*       long source, dest, length;
;*  _map_adr( addr,  space)          -> paddr
;*       long addr; int space;
;*
;*  _map_adr( addr,  -1)             -> addr
;*       sets user seg# from addr
;*
;*  _map_adr( addr,  -2)
;*       control transfer to context at addr.
;*
;*  system call:  mem_cpy
;*       rr6:    source
;*       rr4:    dest
;*       rr2:    length (0 < length <= 64K)
;*        returns
;*       registers unchanged
;*
;*  system call:  map_adr
;*       rr6:    logical addr
;*       r5:     space code
;*       r4:     ignored
;*       rr2:    0
;*        returns
;*       rr6:    physical addr
;*
;*  space codes:
;*       0:      caller data
;*       1:      caller program
;*       2:      system data
;*       3:      system program
;*       4:      TPA data
;*       5:      TPA program
;*
;*       x+256   x=1, 3, 5 : segmented I-space addr.
;*                           instead of data access
;*
;*       FFFF:   set user segment
;*
;************************************************************
```

Listing B-8. (continued)

```
memsc:          ;memory manager system call
                                ; CALLED FROM SC
                                ; IN SEGMENTED MODE
                                ; rr6: source
                                ; rr4: dest   / space
                                ; rr2: length / 0
        testl   rr2
        jr z    mem_map

mem_copy:               ; copy data.
                                ; rr6: source
                                ; rr4: dest
                                ; rr2: length
        ldirb   @r4,@r6,r3
        ldl     rr6,rr4         ; rr6 = dest + length
        ret

mem_map:                ; map address
                                ; rr6: source
                                ; r4:  caller's seg.
                                ; r5:  space
                                ; r2:  caller's FCW
        NONSEG

        cp      r5,#-2          ; space=-2: xfer
        jp  eq  xfersc

        ld      r4,scseg+4(r15)
        ld      r2,scfcw+4(r15)
        calr    map_1
        ldl     cr6+4(r15),rr6  ; return rr6
        SEG
        ret

map_1:                          ; dispatch

        cp      r5,#0FFFFh
        jr eq   set_usr         ; space=-1: user seg

        cpb     rl5,#0
        jr eq   call_data
        cpb     rl5,#1
        jr eq   call_prog
        cpb     rl5,#2
        jr eq   sys_data
        cpb     rl5,#3
        jr eq   sys_prog
        cpb     rl5,#4
        jr eq   usr_data
        cpb     rl5,#5
        jr eq   usr_prog
```

Listing B-8. (continued)

```
        ret                             ;default: no mapping

set_usr:                                ;-1: set user seg.
        ld      _usrseg,r6
        ret

;*
;*** THE FOLLOWING CODE IS SYSTEM-DEPENDENT ***
;*
;*      rr6= logical address
;*      r4 = caller's PC segment
;*      r2 = caller's FCW
;*       returns
;*      rr6= mapped address
;*
;* Most of the system dependencies are in map_prog,
;* which maps a program segment into a data segment
;* for access as data.
;*

call_data:
        bit     r2,#15                  ; segmented caller?
        ret nz                          ; yes-- use passed seg
        ld      r6,r4                   ; no -- use pc segment
        ret                             ; already mapped

call_prog:
        bit     r2,#15                  ; segmented caller?
        jr nz   map_prog                ; yes-- use passed seg
        ld      r6,r4                   ; no -- use pc segment
        jr      map_prog                ; map prog as data

sys_data:
        ld      r6, _sysseg
        ret

sys_prog:
        ld      r6, _sysseg
        ret                             ; assume sys does not
                                        ; separate code, data

usr_data:
        ld      r0, #-1
        cp      r0, _usrseg
        ret eq
        ld      r6, _usrseg
        ret
```

Listing B-8. (continued)

```
usr_prog:
        ld      r0, #-1
        cp      r0, _usrseg
        jr eq   map_prog
        ld      r6, _usrseg
        jr      map_prog

map_prog:               ;map program addr into data
                        ; rr6 = address

        testb   rh5             ; data access?
        ret nz                  ; no: done

        and     r6,#7F00h       ; extract seg bits

        ; olivetti:  segment 8 is the only one with
        ;            separate I and D spaces, and
        ;            the program space is accessed
        ;            as segment 10's data.

        cpb     rh6,#8
        ret ne
        ldb     rh6,#10
        ret
```

Listing B-9. BIOS Trap Handlers

```
;*********** biostrap.8kn cpm.sys + cpmldr.sys ******
;*      Copyright 1984, Digital Research Inc.
;*
;*      Trap handlers for CP/M-8000(tm) BIOS
;*
;* 821013 S. Savitzky (Zilog) -- created
;* 821123 D. Dunlop (Zilog)   -- added Olivetti M20-
;*         specific code to invalidate track buffer
;*         contents when disk drive motor stops
;*         (fixes directory-overwrite on disk change)
;* 830305 D. Sallume (Zilog)  -- added FPE trap
;*         code.
;* 840815 R. WEISER (DRI) -- conditional assembly
;*

__text: .sect

;****************************************************
;*
;* NOTE
;*      Trap and interrupt handlers are started up
;*      in segmented mode.
;*
;****************************************************


;****************************************************
;*
;* Externals
;*
;****************************************************

        .if LOADER
        .global _bios    ; C portion of Loader Bios
        .else
        .global __bios   ;C portion of Normal Bios
        .endif
        .global memsc            ;memory-management SC
        .global _tbvalid         ;disk track buff valid
        .global _tbdirty         ;disk track buff is dirty

        .global _sysseg, _usrseg, _sysstk, _psap,
        .if LOADER
        .else                    ; only the normal Bios
        .global fp_epu
        .endif
```

Listing B-9. (continued)

```
;*********************************************************
;*
;* M-20 ROM scratchpad RAM addresses
;*
;*********************************************************

rtc_ext:  .equ  82000022h       ;Place to put address
                                ; of list of functions
                                ; for each clock tick
motor_on: .equ  82000020h       ;Disk motor timeout

;*********************************************************
;*
;* Global declarations
;*
;*********************************************************

        .global _trapinit
        .global _trapvec
        .global _trap

        .global xfersc


;*********************************************************
;*
;* System Call and General Trap Handler And Dispatch
;*
;*      It is assumed that the system runs
;*      non-segmented on a segmented CPU.
;*
;*      _trap is jumped to segmented, with the
;*      following information on the stack:
;*
;*              trap type: WORD
;*              reason:    WORD
;*              fcw:       WORD
;*              pc:        LONG
;*
;*      The trap handler is called as a subroutine,
;*      with all registers saved on the stack,
;*      IN SEGMENTED MODE.  This allows the trap
;*      handler to be in another segment (with some
;*      care).  This is useful mainly to the debugger.
;*
;*      All registers except rr0 are also passed
;*      intact to the handler.
;*
;*********************************************************
```

Listing B-9. (continued)

```
__text: .sect

sc_trap:                        ;system call trap server

        push    @r14,@r14


_trap:

        sub     r15,#30         ; push caller state
        ldm     @r14,r0,#14
        NONSEG                  ; go nonsegmented
        ldctl   r1,NSP
        ld      nr14(r15),r14
        ex      r1,nr15(r15)

                                ; trap# now in r1
        cpb     rh1,#7Fh        ; system call?
        jr  ne  trap_disp       ;       no
                                ;       yes: map it
        clrb    rh1
        add     r1,#SC0TRAP

;=== need range check ===

trap_disp:                      ; dispatch
        sll     r1,#2
        ldl     rr0,_trapvec(r1)
        testl   rr0
        jr  z   _trap_ret       ; zero -- no action
                                ; else call seg @rr0
        pushl   @r15,rr0        ;  (done via kludge)
        SEG
        popl    rr0,@r14
        calr    trap_1
        jr      _trap_ret

trap_1:                         ; jp @rr0
        pushl   @r14,rr0
        ret
```

Listing B-9. (continued)

```
_trap_ret:                  ;return from trap or interrupt

         NONSEG
         ld       r1,nr15(r15)      ; pop state
         ld       r14,nr14(r15)
         ldctl    NSP,r1
         SEG                        ; go segmented for the iret.
         ldm      r0,@r14,#14
         add      r15,#32

         iret                       ; return from interrupt


;*************************************************************
;*
;* Assorted Trap Handlers
;*
;*************************************************************

epu_trap:
         push     @r14,#EPUTRAP
         jr       _trap

pi_trap:
         push     @r14,#PITRAP
         jr       _trap

seg_trap:
         push     @r14,#SEGTRAP
         jr       _trap

nmi_trap:
         push     @r14,#NMITRAP
         jr       _trap

         .if LOADER
         .else              ; not used in Loader Bios
```

Listing B-9. (continued)

```
;*****************************************************
;*
;* Bios system call handler
;*
;*****************************************************


biossc:                         ;call bios
        NONSEG
                                ; r3 = operation code
                                ; rr4= P1
                                ; rr6= P2

        ld      r0,scfcw+4(r15) ; if caller nonseg, normal
        and     r0,#0C000h
        jr  nz  seg_ok

        ld      r4,scseg+4(r15) ; then add seg to P1, P2
        ld      r6,r4
seg_ok:
                                ; set up C stack frame
;====
        pushl   @r15,rr6
        pushl   @r15,rr4
        push    @r15,r3

                                ; call C program
        call    __bios

                                ; clean stack & return
        add     r15,#10
        ldl     cr6+4(r15),rr6  ; with long in rr6

        SEG
        ret

        .endif
```

Listing B-9. (continued)

```
;*********************************************************
;*
;* Context Switch System Call
;*
;*      xfer(context)
;*      long context;
;*
;* context is the physical (long) address of:
;*      r0
;*      ...
;*      r13
;*      r14 (normal r14)
;*      r15 (normal r15)
;*      ignored word
;*      FCW (had better specify normal mode)
;*      PC segment
;*      PC offset
;*
;* The system stack pointer is not affected.
;*
;* Control never returns to the caller.
;*
;*********************************************************

xfersc:                         ;enter here from system call
        SEG

; build frame on system stack

; when called from system call, the frame replaces
; the caller's context, which will never be resumed.

        inc     r15,#4          ;discard return addr
        ldl     rr4,rr14        ;move context
        ld      r2,#FRAMESZ/2
        ldir    @r4,@r6,r2
        jr      _trap_ret       ;restore context
```

Listing B-9. (continued)

```
;*****************************************************
;*
;*  _motor_c -- check if disk motor still running.
;*              Entered each clock tick.  Invalidates
;*              track buffer when motor stops
;*              (Note: runs segmented)
;*
;*****************************************************

_motor_c:
        ldl     rr4,#motor_on   ;Motor running?
        test    @r4
        ret     nz              ;Yes: do nothing
        ldar    r4,$
        ld      r5,#_tbdirty    ; Is track buff dirty?
        test    @r4             ; Yes...
        ret     nz              ;   ...return without invalidating
        ld      r5,#_tbvalid
        clr     @r4             ;No: mark track buffer
        ret                     ; invalid

; Table of functions run each real time clock tick

_ticktab:
        .long   -1              ;Will contain _motor_c
        .word   0ffffh          ;Terminator

;*****************************************************
;*
;*  _trapinit -- initialize trap system
;*
;*****************************************************


;*
;*  PSA (Program Status Area) structure
;*
ps      .equ    8       ; size of a program status entry
                        ; --- segmented ---
```

Listing B-9. (continued)

```
psa_epu .equ    1*ps      ; EPU trap offset
psa_prv .equ    2*ps      ; priviledged instruction trap
psa_sc  .equ    3*ps      ; system call trap
psa_seg .equ    4*ps      ; segmentation trap
psa_nmi .equ    5*ps      ; non-maskable interrupt
psa_nvi .equ    6*ps      ; non-vectored interrupt
psa_vi  .equ    7*ps      ; vectored interrupt
psa_vec .equ    psa_vi+(ps/2)    ; vectors


_trapinit:

; initialize trap table

        lda     r2,_trapvec
        ld      r0,#NTRAPS
        subl    rr4,rr4
clrtraps:
        ldl     @r2,rr4
        inc     r2,#4
        djnz    r0,clrtraps

        ld      r2,_sysseg
        .if LOADER
        .else             ;not used by Loader Bios
        lda     r3,biossc
        ldl     _trapvec+(BIOS_SC+SC0TRAP)*4,rr2
        .endif
        lda     r3,memsc
        ldl     _trapvec+(MEM_SC+SC0TRAP)*4,rr2
        .if LOADER
        .else             ; not used by Loader Bios
        lda     r3,fp_epu
        ldl     _trapvec+EPUTRAP*4,rr2
        .endif

; initialize some PSA entries.
;       rr0     PSA entry: FCW  (ints ENABLED)
;       rr2     PSA entry: PC
;       rr4     -> PSA slot

        ldl     rr4,_psap
        SEG
        ldl     rr0,#0000D800h  ; traps here
```

Listing B-9. (continued)

```
        add     r5,#ps                  ; EPU trap
        ldar    r2,epu_trap
        ldm     @r4,r0,#4

        add     r5,#ps                  ; Priviledged Inst
        ldar    r2,pi_trap
        ldm     @r4,r0,#4

        add     r5,#ps                  ; System Call
        ldar    r2,sc_trap
        ldm     @r4,r0,#4

        add     r5,#ps                  ; segmentation
        ldar    r2,seg_trap
        ldm     @r4,r0,#4

        add     r5,#ps                  ; Non-Maskable Int.
        ldar    r2,nmi_trap
        ldm     @r4,r0,#4

; Set up Real-Time Clock external call loc

        ldar    r2,_motor_c
        ldar    r4,_ticktab
        ldl     @r4,rr2
        ldl     rr2,#rtc_ext
        ldl     @r2,rr4
        NONSEG
        ret

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

Listing B-10. System-Call Interface

```
;********** syscall.8kn cpm.sys + cpmldr.sys ********
;*      Copyright 1984, Digital Research Inc.
;*
;*      System Call interface for CP/M-8000(tm) BIOS
;*
;* 820927 S. Savitzky (Zilog) -- created.
;* 840815 R. Weiser (DRI) -- conditional assembly
;*

__text: .sect

;*************************************************
;*
;* NOTE
;*      The following system call interface routines
;*      are designed to be called from non-segmented
;*      C programs.
;*
;*      Addresses are passed as LONGs.
;*
;*************************************************

        .global _xfer
        .global _mem_cpy
        .global _map_adr
        .global _bios
        .global _bdos
```

Listing B-10. (continued)

```
;*****************************************************
;*
;*  Context Switch Routine
;*
;*       xfer(context)
;*       long context;
;*
;*  context is the physical (long) address of:
;*       r0
;*       ...
;*       r13
;*       r14 (normal r14)
;*       r15 (normal r15)
;*       ignored word
;*       FCW (had better specify normal mode)
;*       PC segment
;*       PC offset
;*
;*  The system stack pointer is not affected.
;*
;*  Control never returns to the caller.
;*
;*****************************************************

_xfer:

        ldl     rr6,ARG1(r15)
        ldl     rr4,#-2
        subl    rr2,rr2
        sc      #XFER_SC
        ret
```

Listing B-10. (continued)

```
;*********************************************************
;*
;* System/User Memory Access
;*
;* _mem_cpy( source, dest, length)
;*      long source, dest, length;
;* _map_adr( addr,  space)        -> paddr
;*      long addr; int space;
;*
;* _map_adr( addr, -1)
;*      sets user segment # from addr.
;*
;* _map_adr( addr, -2)
;*      transfer to context block at addr
;*
;* system call:  mem_cpy
;*      rr6:    source
;*      rr4:    dest
;*      rr2:    length (0 < length <= 64K)
;*       returns
;*      registers unchanged
;*
;* system call:  map_adr
;*      rr6:    logical addr
;*      r5:     space code
;*      r4:     ignored
;*      rr2:    0
;*       returns
;*      rr6:    physical addr
;*
;* space codes:
;*      0:      caller data
;*      1:      caller program
;*      2:      system data
;*      3:      system program
;*      4:      TPA data
;*      5:      TPA program
;*
;*      x+256   return segmented instruction address,
;*               not data access address
;*
;*      FFFF    set user-space segment from address
;*
;*********************************************************
```

Listing B-10. (continued)

```
_mem_cpy:                       ;copy memory C subroutine
;====
        ldl     rr6,ARG1(r15)
        ldl     rr4,ARG3(r15)
        ldl     rr2,ARG5(r15)
        sc      #MEM_SC
        ret

_map_adr:                       ;map address C subroutine
;====
        ldl     rr6,ARG1(r15)
        ld      r5, ARG3(r15)
        subl    rr2,rr2                 ; 0 length says map
        sc      #MEM_SC
        ret

        .if LOADER
        .else                   ; not used by Loader Bios

;****************************************************
;*
;* long _bios(code, p1, p2)
;* long _bdos(code, p1)
;*      int code;
;*      long p1, p2;
;*
;* BIOS, BDOS access
;*
;****************************************************

_bios:
        ld      r3,ARG1(r15)
        ldl     rr4,ARG2(r15)
        ldl     rr6,ARG4(r15)
        sc      #BIOS_SC
        ret

_bdos:
        ld      r5,ARG1(r15)
        ldl     rr6,ARG2(r15)
        sc      #BDOS_SC
        ret

        .endif
```

End of Appendix B

APPENDIX C
PUTBOOT Utility C Language Source

Listing C-1. Bootstrap Writer for the Olivetti M20

```
/*=======================================================================*/
/*+---------------------------------------------------------------------+*/
/*|                                                                     |*/
/*|     CP/M-Z8K(tm) Bootstrap Writer for the OLIVETTI M20 (Z8000)      |*/
/*|                                                                     |*/
/*|     Copyright 1984, Digital Research Inc.                           |*/
/*|                                                                     |*/
/*+---------------------------------------------------------------------+*/
/*=======================================================================*/

char *copyrt = "CP/M-Z8K(tm) Ver. 1.1, Copyright 1984, Digital Research Inc.";
char *serial = "XXXX-0000-654321";


/* HISTORY
**
**      830801  F. Zlotnick (Zilog) -- written
**      840524  rfw modified includes
**      840801  rfw made to look generic
*/

#include "portab.h"
#include "osif.h"                       /* cpm.h and bdos.h replaces with    */
#include "stdio.h"                      /* osif.h  03-15-84 rfw              */
#include "bdos.h"
#include "xout.h"

#define CDATA   0                       /* Parameter for map_adr()           */
#define DIRSEC  1                       /* Parameter for BIOS Write call     */

#define SETTRK  10                      /* BIOS Function 10 = Set Track      */
#define SETSEC  11                      /* BIOS Function 11 = Set Sector     */
#define BSETDMA 12                      /* BIOS Function 12 = Set DMA Addr   */
#define WSECTOR 14                      /* BIOS Function 14 = Write Sector   */

XADDR   physdir;                        /* Segmented address of dirbuf       */

struct  dpbs    idpb;                   /* Disk Parameter Block              */
struct  bios_parm ibp;                  /* BIOS param block for BDOS call 50*/
XADDR   physibp;                        /* physical address of ibp structure*/

extern  long    map_adr();              /* Function to return physical addr */
```

Listing C-1. (continued)

```
#define BPLS     128                        /* Bytes per logical sector */
#define BPS      256                        /* Bytes per sector */
#define BPSO     128                        /* Bytes per sector, trk 0 */
#define SPT      16                         /* sectors per track */
#define LSPT     32                         /* Logical sectors per track */
#define SYSTRKS  2                          /* Number of boot tracks */
#define SYSSIZE  SPT*BPS*SYSTRKS            /* Max size of bootstrap */
#define STARTRK  1                          /* Track number to start on */

FILE     *fin;

char     syscode[SYSSIZE];                  /* Hold the entire bootstrap here! */
char     *system = "CPMLDR.SYS";            /* Name of the prog to boot */


struct   x_hdr   xh;
struct   x_sg    xs;
int      dsknum;                   /* Drive number 0-15 = A-P  */

main(argc,argv)
int      argc;
char     *argv[];

         register int     i, j, c;
         register char    *p;
         long     fsize;
         int      curdsk;                            /* Good to remember, & reset*/

         if(argc != 3) usage();
         system = *++argv;
         if( (dsknum = **++argv - 'a') < 0 || dsknum > 15) {
                 printf("putboot: Illegal drive code %cn", *argv[0]);
                 exit(1);
         }
         curdsk = _ret_cdisk();
         _get_dpb(map_adr((long) &idpb, CDATA)); /* Physaddr of idpb */
         if( (fin = fopenb(system, "r")) == NULL) {
                 printf("putboot: Can't open bootstrap file %sn", system);
                 exit(1);
         }
         fsize = 0L;
```

Listing C-1. (continued)

```
                /* read file header */

        p = (char *) &xh;
        for (i = 0; i < sizeof(xh); i++)
                *p++ = (char) getc(fin);

                /* read and count segment headers to get file size */
        for (i = 0; i < xh.x_nseg; i++) {
                p = (char *) &xs;
                for( j = 0; j < sizeof(xs); j++)
                        *p++ = (char) getc(fin);
                if( xs.x_sg_typ != X_SG_BSS && xs.x_sg_typ != X_SG_STK)
                        fsize += xs.x_sg_len;
        }
        if (fsize > SYSSIZE) {
                printf("Bootstrap too bign");
                exit(1);
        }
        else
                printf("Bootstrap file is %ld bytesn", fsize);
        p = syscode;

/* If any other special information is needed at the beginning of the */
/* system track of the loader load them into the syscode area now.    */
        while(fsize--) {
                if( (c = getc(fin)) == EOF) {
                        printf("Unexpected EOF in %s, %ld leftn",system,fsize);
                        exit(1);
                }
                *p++ = c;
        }

/*
** At this point, the entire bootstrap program code and data has been loaded
**  into the array named "syscode", preceded by a bunch of PCOS garbage
**  which the Olivetti boot PROM expects to find there.  Now we use direct
**  BIOS calls to write the syscode array out to the proper area on disk.
**  For the Olivetti, this is tracks 1 and 2, since track 0 is special.
*/

        putboot(syscode);
        printf("Bootstrap has been written.n");
        _sel_disk(curdsk);                      /* reselect original disk*/
}
```

Listing C-1. (continued)

```
putboot(code)
char    *code;
{
        register int    i;                      /* Handy index                 */
        register int    nlsecs;                 /* # logical sectors in boot*/
        register char   *p;                     /* ptr to next part of code */

        int     track;                          /* Current track               */
        int     sector;                         /* Current sector              */

        physibp = map_adr( (long) &ibp, CDATA );
        nlsecs = SYSTRKS * LSPT;                /* size / log secs per trk  */

                        /* Pause for the user to insert disk */
/*      pause(drvname); */

/* Put code here when trk0 sect0 are special */
/* do a _sel_disk for the drive you want */
/* then call putblk trk0, sect0, and the address of sector information */
        p = code;
        for(i = 0; i < nlsecs; i++) {
                track = STARTRK + i/LSPT;
                sector = i%LSPT;
                putblk(track, sector, p);
                p += BPLS;
        }
}

/*
 * Function to select a given track for writing on, on the current disk.
 * Makes use of the BDOS direct BIOS call to issue Bios function 10.
 */
settrk(n)
int     n;
{
        ibp.req = SETTRK;                 /* BIOS request number 10 */
        ibp.p1 = (long) n;                /* parameter = track # */
        _bios_call( physibp );            /* Pass seg ibp address */
```

Listing C-1. (continued)

```
/*
 * Function to put block i of the boot track.
 */
putblk(trk, sec, addr)
int     trk, sec;
char    *addr;
{
        register int    n;

        _sel_disk(dsknum);              /* select as current disk */
        settrk(trk);

        n = sec + 1;                    /* sector number */
        ibp.req = SETSEC;               /* BIOS request number 11 */
        ibp.p1 = (long) n;              /* parameter = sector # */
        _bios_call( physibp );          /* Pass seg ibp address */

                /* Sector is now set; now set dma address. */

        ibp.req = SSETDMA;              /* BIOS Request number 12 */
        ibp.p1 = map_adr( (long) addr, CDATA);
                                        /* param = seg address of I/O buffer */
        _bios_call( physibp );          /* Call BIOS */

                /* Now can do a write */

        ibp.req = WSECTOR;              /* BIOS Request number 14 */
        ibp.p1 = DIRSEC;                /* Complete write immediately */
        _bios_call( physibp );          /* Do it! */
}

/*
 * If the user invoked us with the wrong number of args...
 */
usage()
{
        printf("Usage: putboot <filename> <drivecode>n");
        exit(1);
}
```

End of Appendix C

Index