Donn Stewart
13917 Deviar Dr
Centreville, VA 20120
dstew@cpuville.com

Designing, Building, and Selling Obsolete Computers -- for Educational Purposes -- since 2004

Setting up CP/M 2.2 on a New Z80 Computer

This is to document how I set up CP/M 2.2 to run on a newly created Z80-based microcomputer, intended as a kit for hobbyists or educators. The computer was designed to be extremely simple to build and understand, and as such does not implement interrupts or direct memory access, even though the Z80 can support these functions. My goal was to set up CP/M in the simplest way possible.

The System Architecture

Memory Space

The computer system memory space takes on one of two configurations, software selected through OUT instructions to port 0 or port 1. In configuration 0, memory locations 0x0000 to 0x07FF are ROM, and locations 0x0800 to 0xFFFF are RAM. In configuration 1, the memory is all RAM from 0x0000 to 0xFFFF. This dual configuration is needed to solve the issue presented by the Z80 CPU's requirement that some code be present at location 0x0000 when it is started, and CP/M's requirement that location 0x0000 be available for RAM. There are other ways to solve this problem of course, but with 2K of ROM present in configuration 0, subroutines can be stored there that will aid in bringing up the CP/M system.

Input/Output Ports

As mentioned above, ports 0 and 1 are used to select the memory configuration. An OUT (0),A instruction selects configuration 0 (2K ROM + 62K RAM), and an OUT (1),A instruction selects configuration 1 (all-RAM). There is no data transferred by these instructions, and there are no input ports implemented at addresses 0 or 1.

There is a single serial interface implemented that uses port addresses 2 and 3. Port 2 is the data input and output port, and port 3 is the UART status/control port. The port is initialized at system start (or system reset) by code in the ROM. It does not have to be initialized by CP/M.

The single IDE disk interface uses input/output ports 8 through 15. Only the IDE command block registers are implemented; the control block registers, which are selected if CS3FX- is asserted, are not used (CS3FX- is tied to VCC). They are not needed for proper function of the disk interface. The disk is configured so that input and output data bits 0 though 7 are sent to the system data bus; bits 8 through 15 are ignored. This means that half the storage on the drive is not used, but it simplifies the hardware and software drive interface.

The Software

The ROM System Monitor

The 2K ROM contains a system monitor program with several functions accessible from a simple command line interface, or as subroutines called by user programs. The ROM monitor assembly language file is here, and the list file is here. There are commands to examine memory (dump), change memory (load), and run programs (run). There are commands to dump binary data from the memory to the serial port (bdump) and to load binary data from the serial port (bload). There are commands to read a disk sector and place it in memory (diskrd), and to write a sector to disk from memory (diskwr). Finally, there is a command to start CP/M (cpm), assuming it has been installed.

The RAM System Monitor

I made a copy of the ROM system monitor, assembled to address 0xDC00, so that it sits just below CP/M in the all-RAM memory configuration. This RAM monitor (the assembly language file is here and the listing is here) is for use when ROM-type functions are needed with the computer memory in configuration 1 (all-RAM). It is needed for getting the first few programs into the CP/M system through the serial port. I also wrote a CP/M version of the RAM monitor, MONITOR.COM, that was assembled to location 0x0100. The MONITOR assembly language file is here, and the listing is here. This program, when run, copies itself from location 0x0100 to location 0xDC00, to be clear of the 0x0100 address area where CP/M programs are loaded and run. It is useful for further loading of programs into CP/M by the serial port, without having to reset the computer, once CP/M is up and running.

CP/M 2.2

Source Code and Manuals

CP/M is a very flexible operating system. It uses a standard core of system software that is almost entirely machine independent, other than the requirement for an 8080 or Z80 CPU. The standard core consists of the Console Command Processor (CCP) and the Basic Disk Operating System (BDOS). In addition to the standard, machine-independent core, there is a machine-dependent Basic Input/Output System (BIOS). The BIOS contains the drivers for the various hardware elements used by CP/M to operate the disk system and communicate through the input and output ports. It also contains the disk parameter tables that are used by CP/M to create and use the file system. The main task in creating a CP/M system for a new computer is to write a customized BIOS tailored to that new computer's architecture.

The CP/M source code I used is found at The Unofficial CP/M Web Site. The archive CP/M 2.2 ASM SOURCE on the Digital Research Sources page on that site was used to

build this system. The source file CPM22.Z80 was the one I assembled. This source contains the CCP and BDOS components of CP/M, along with a skeleton of the BIOS jump table (needed to assemble correctly). This source file is the work of Clark A. Calkins, and was obtained by disassembling a working system. The other CP/M sources on the web site were obtained from the original Digital Research documents by scanning and optical character recognition (OCR). As such, they have a few OCR-type errors, and are also in 8080 rather than Z80 assembly language. One can convert the 8080 to Z80 by using the awk script toZ80 (found at Github here). The script is not perfect, but most of the problems with translation are easy to recognize and fix. Nevertheless, just to get CP/M up and running, I used the CPM22.Z80 source.

The CP/M 2.0 System Manual has been put on this web page: www.gaby.de/cpm/manuals/archive/cpm22htm/. This manual describes CP/M for the user, including the basic operating system commands. An OCR-derived CP/M 2.0 Alteration Guide document is present here also as chapter 6 of the System Manual. A scan of the original Alteration Guide is available here. The Alteration Guide explains how to create a new CP/M 2 system. However, it is targeted to a system with floppy drives, and assumes that one already has a CP/M 2 system up and running, and can use CP/M commands such as MOVECPM and DDT (the debugger program). I did not have this available. (Yes, I could have used one of the fine CP/M emulators out there, but I wanted to do everything from scratch.) However, the Alteration Guide was useful in a number of ways, as I will explain below. Finally, there is the CP/M 2.0 Interface Guide, which describes how to use CP/M features in user programs. A scanned copy of the Interface Guide can be found here.

CCP and BDOS Assembly

Only a few small changes were needed to assemble the CPM22.Z80 source for a 64K system. I set the code origin for the CCP to 0xE400. I adjusted a few zeros after the CCP to put the the BDOS entry at 0xEC06, and similarly at the end of the BDOS to make the BIOS entry at 0xFA00. These are the standard entry points for the CCP, BDOS, and BIOS according to the CP/M 2.0 Alteration Guide. I named the binary output file of this assembly cpm22.sys.

Creating the BIOS

I used the skeletal customizable BIOS code found as an appendix to the CP/M Alteration Guide as the basis for a custom BIOS for this computer. This source was only available in 8080 code, so I had to translate it to Z80 code using the AWK script toZ80 as described above. There was one OCR error in the source, an 8080 DAD 0 instruction toward the end of the SELDSK subroutine that should have been DAD D, otherwise the code was correct. I had to fix a few translation errors as well. Using this source as the basis, I wrote the minimum amount of code needed to get CP/M working, as described below. The finished customized BIOS assembly language is here, and the listing is here.

The BIOS Coding Tasks

There are several coding tasks needed to create a functional BIOS tailored for a certain computer system. One needs to write cold and warm boot routines, drivers for the input/output ports, drivers for the disk drive(s), and disk parameter tables.

Boot Routines and Disk Parameter Tables

The cold and warm start boot routines in the skeletal BIOS are written for the classical standard system with four 8-inch IBM floppy drives. I wanted to make as few changes to the skeletal BIOS source code as possible, so I left these subroutines intact. I also left the disk parameter tables essentially unchanged. See below for details on how CP/M, which thinks it is using four floppy disks, ends up using a 1 Gb hard disk.

Coding for the Console Input and Output

The CP/M Alteration Guide has fairly detailed explanations for what each BIOS subroutine requires, that is, which registers are used to convey information to the subroutine, and how data and error codes should be returned. The subroutines for console input and output are very simple, just single character output (CONOUT), single character input (CONIN), and input status query (CONST). Here is the code I wrote for these three subroutines:

```
conin:                  ;returns console character in register a
        in    a,(3)     ;get status
        and   002h      ;check RxRDY bit
        jp    z,conin   ;loop until char ready
        in    a,(2)     ;get char
        AND 7fh         ;strip parity bit
        ret
;
conout:                 ;console character output from register c
        in    a,(3)
        and   001h      ;check TxRDY bit
        jp    z,conout  ;loop until port ready
        ld    a,c       ;get the char
        out   (2),a     ;out to port
        ret
;
const:                  ;console status, return 0ffh if character ready, 00h if not
        in    a,(3)     ;get status
        and   002h      ;check RxRDY bit
        jp    z,no_char
        ld    a,0ffh    ;char ready
        ret
no_char: ld   a,00h     ;no char
        ret
```

There are subroutines in the BIOS for character output to a list device (a printer), and for output to a punch device, and input from a reader device. The punch and reader subroutines were intended as drivers for a paper tape puncher and reader. These other character input and output drivers can be used for a variety of physical devices, but since my system has only one serial port, which is dedicated to the console, I simply left these drivers as they were in the skeletal BIOS -- simple returns without performing any actions.

Coding for the Disk System

The remainder of the subroutines in the BIOS have to do with disk input and output. As mentioned above, CP/M comes out of the era when essentially all systems used floppy disks. Floppy disk controllers are fairly complicated compared to modern IDE drives. Most were run using interrupt-driven, direct-memory access hardware, and often used subroutines present in a system ROM. These drives were typically much slower than the computer system, and there were timing issues to be dealt with in the driver software. Disk reading and writing errors were very common, and the software had to deal with these. The disk in a particular drive could be changed, and the software had to deal with this possibility also. So there is a lot of code in old BIOSes that is dedicated to running these floppy disks. An example of this complexity can be seen in a standard CP/M 2.2 BIOS here:
www.gaby.de/cpm/manuals/archive/cpm22htm/axa.htm

However, with the new Z80 system described here, an IDE disk interface to a hard disk is used instead. The disk interface is very simple. The disk is much faster than the system CPU, so there are no timing issues to worry about. Modern hard disks are essentially error-free. The disk cannot be changed. This makes the software to operate the disk very simple to write.

Subroutines Needed for Reading and Writing the Disk

SETDMA, SELDSK, SETTRK, and SETSEC Subroutines

The CP/M BDOS accesses the disk by setting the memory address for a 128-byte sector buffer (SETDMA subroutine), selecting the disk to be accessed (SELDSK), and setting the track (SETTRK), and sector (SETSEC) to be accessed. In old BIOSes, especially those written for disk hardware using interrupts and direct memory access, these subroutines interacted directly with the disk hardware. Once these settings were made, a disk read or write command was given, and the software waited for the task to be finished. However, with the IDE disk, these subroutines need merely set variables in RAM to be used by the disk read and write subroutines. The SELDSK subroutine has additional code that returns the address to the proper disk parameter table header. More about the disk parameter tables later.

```
;
setdma:                 ;set dma address given by registers b and c
        LD    l, c      ;low order address
        LD    h, b      ;high order address
        LD    (dmaad),HL ;save the address
        ret
;
seldsk:                 ;select disk given by register c
        LD    HL, 0000h ;error return code
        LD    a, c
        LD    (diskno),A
        CP    disks     ;must be between 0 and 3
        RET   NC        ;no carry if 4, 5,...
;                       disk number is in the proper range
;       defs  10        ;space for disk select
;                       compute proper disk Parameter header address
        LD    A,(diskno)
```

```
        LD    l, a          ;l=disk number 0, 1, 2, 3
        LD    h, 0          ;high order zero
        ADD   HL,HL         ;*2
        ADD   HL,HL         ;*4
        ADD   HL,HL         ;*8
        ADD   HL,HL         ;*16 (size of each header)
        LD    DE, dpbase
        ADD   HL,DE         ;hl=,dpbase (diskno*16). Note typo "DAD 0" here in
                            original 8080 source.
        ret
;
settrk:                     ;set track given by register c
        LD    a, c
        LD    (track),A
        ret
;
setsec:                     ;set sector given by register c
        LD    a, c
        LD    (sector),A
        ret
;
```

Disk READ and WRITE Subroutines

An important issue with CP/M 2.2 is the requirement for disk data to be handled in 128-byte pieces. This comes out of the floppy disk era when disk sectors were this size. Modern disks have larger sectors, 512 bytes in the IDE standard (although the hardware of this computer only uses 256 bytes of each sector). CP/M 2.2 has some available BIOS routines for sector deblocking, so that sector sizes larger than 128 bytes can be used, the software breaking up the larger sector into the 128-byte pieces that CP/M wants to see. We could use the CP/M sector deblocking code to make use of the full 256 bytes available in each sector, but with a 1 Gb hard drive, we do not need to preserve disk space. Therefore, in the interest of simplicity and understandability, I decided to use only 128 bytes of each sector. Yes, that means I am only using one-quarter of the available disk space, but this is still a vast disk space for an 8-bit system. The disk READ and WRITE subroutines below use a 256-byte host buffer for the physical access, and additional code sends to or receives from the 128-byte buffer set by the SETDMA subroutine the top 128 bytes of the 256 byte physical sector.

Modern hard disks can use either cylinder-head-sector (CHS) addressing or logical block addressing (LBA). In LBA mode, the disk is presented as one long array of sectors, numbered from 0x000000 to 0x1FFFFF for a 1 Gb disk. I decided to use LBA to access the disk in a very simple way. The BIOS disk read and write subroutines set LBA bits 0 to 7 to the CP/M sector, bits 8 to 15 to the CP/M track, and bits 16 and 17 to the CP/M disk. For this scheme to work one needs at least a 340 Mb drive, but drives of this capacity are easily found. To use a disk of smaller capacity one would need to write code to translate the CP/M addressing to LBA addressing more efficiently, in a way that did not waste so much space. But, just to get it working, this is what I did.

Here are the disk READ and WRITE subroutines for my customized BIOS:

```
read:
;Read one CP/M sector from disk.
;Return a 00h in register a if the operation completes properly, and 01h if an error occurs during the read.
;Disk number in 'diskno'
;Track number in 'track'
;Sector number in 'sector'
;Dma address in 'dmaad' (0-65535)
;
                        ld      hl,hstbuf               ;buffer to place disk sector (256 bytes)
rd_status_loop_1:       in      a,(0fh)                 ;check status
                        and     80h                     ;check BSY bit
                        jp      nz,rd_status_loop_1     ;loop until not busy
rd_status_loop_2:       in      a,(0fh)                 ;check status
                        and     40h                     ;check DRDY bit
                        jp      z,rd_status_loop_2      ;loop until ready
                        ld      a,01h                   ;number of sectors = 1
                        out     (0ah),a                 ;sector count register
                        ld      a,(sector)              ;sector
                        out     (0bh),a                 ;lba bits 0 - 7
                        ld      a,(track)               ;track
                        out     (0ch),a                 ;lba bits 8 - 15
                        ld      a,(diskno)              ;disk (only bits 16 and 17 used)
                        out     (0dh),a                 ;lba bits 16 - 23
                        ld      a,11100000b             ;LBA mode, select host drive 0
                        out     (0eh),a                 ;drive/head register
                        ld      a,20h                   ;Read sector command
                        out     (0fh),a
rd_wait_for_DRQ_set:    in      a,(0fh)                 ;read status
                        and     08h                     ;DRQ bit
                        jp      z,rd_wait_for_DRQ_set   ;loop until bit set
rd_wait_for_BSY_clear:  in      a,(0fh)
                        and     80h
                        jp      nz,rd_wait_for_BSY_clear
                        in      a,(0fh)                 ;clear INTRQ
read_loop:              in      a,(08h)                 ;get data
                        ld      (hl),a
                        inc     hl
                        in      a,(0fh)                 ;check status
                        and     08h                     ;DRQ bit
                        jp      nz,read_loop            ;loop until clear
                        ld      hl,(dmaad)              ;memory location to place data
```

```
                                                read from disk
                        ld      de,hstbuf       ;host buffer
                        ld      b,128           ;size of CP/M sector
rd_sector_loop:         ld      a,(de)          ;get byte from host buffer
                        ld      (hl),a          ;put in memory
                        inc     hl
                        inc     de
                        djnz    rd_sector_loop  ;put 128 bytes into memory
                        in      a,(0fh)         ;get status
                        and     01h             ;error bit
                        ret
write:
;Write one CP/M sector to disk.
;Return a 00h in register a if the operation completes properly, and 0lh if an error
occurs during the read or write
;Disk number in 'diskno'
;Track number in 'track'
;Sector number in 'sector'
;Dma address in 'dmaad' (0-65535)
                        ld      hl,(dmaad)      ;memory location of data to
                                                write
                        ld      de,hstbuf       ;host buffer
                        ld      b,128           ;size of CP/M sector
wr_sector_loop:         ld      a,(hl)          ;get byte from memory
                        ld      (de),a          ;put in host buffer
                        inc     hl
                        inc     de
                        djnz    wr_sector_loop  ;put 128 bytes in host buffer
                        ld      hl,hstbuf       ;location of data to write to
                                                disk
wr_status_loop_1:       in      a,(0fh)         ;check status
                        and     80h             ;check BSY bit
                        jp      nz,wr_status_loop_1 ;loop until not busy
wr_status_loop_2:       in      a,(0fh)         ;check status
                        and     40h             ;check DRDY bit
                        jp      z,wr_status_loop_2 ;loop until ready
                        ld      a,01h           ;number of sectors = 1
                        out     (0ah),a         ;sector count register
                        ld      a,(sector)
                        out     (0bh),a         ;lba bits 0 - 7 = "sector"
                        ld      a,(track)
                        out     (0ch),a         ;lba bits 8 - 15 = "track"
                        ld      a,(diskno)
                        out     (0dh),a         ;lba bits 16 - 23, use 16 to 20
                                                for "disk"
                        ld      a,11100000b     ;LBA mode, select drive 0
```

```
                        out     (0eh),a                 ;drive/head register
                        ld      a,30h                   ;Write sector command
                        out     (0fh),a
wr_wait_for_DRQ_set:    in      a,(0fh)                 ;read status
                        and     08h                     ;DRQ bit
                        jp      z,wr_wait_for_DRQ_set   ;loop until bit set
write_loop:             ld      a,(hl)
                        out     (08h),a                 ;write data
                        inc     hl
                        in      a,(0fh)                 ;read status
                        and     08h                     ;check DRQ bit
                        jp      nz,write_loop           ;write until bit cleared
wr_wait_for_BSY_clear:  in      a,(0fh)
                        and     80h
                        jp      nz,wr_wait_for_BSY_clear
                        in      a,(0fh)                 ;clear INTRQ
                        and     01h                     ;check for error
                        ret
;
```

The label hstbuf points to a 256-byte area in RAM that holds the data after transfer from or before transfer to the disk. One might note the use of polling status bits to decide when all 256 bytes of data have been transferred. One can code ones own 256-byte counter, or use the 256-byte counter that is internal to the drive. I used the latter method here.

Disk Parameter Tables

CP/M needs a logical description of the disk system in order to operate properly. This description is needed by the system software that creates directories and files, and watches for available disk space. This description is held in a series of disk parameter tables in the BIOS, described in more detail below when I discuss the SECTRAN subroutine. The disk parameter tables do not need to be a true description of the actual physical disk drive(s), as long as one creates in the disk read and write subroutines a way to translate a BDOS request to read or write a particular logical disk sector into a read or write of a unique physical disk sector. The standard CP/M system, which is represented in the unmodified skeletal BIOS, has four 8" IBM floppy disks, each with 77 26-sector tracks (numbered 1 to 26), and a total capacity of about 250 Kb. This disk setup is also coded into the BIOS cold and warm start routines, which load the operating system from the disk into memory. Again, I decided to stick with this original, standard logical disk description, so that I was making the fewest changes to the skeletal BIOS as possible. Here are the disk parameter tables in my finished BIOS:

```
;
; fixed data tables for four-drive standard
; ibm-compatible 8" disks
; no translations
;
```

```
; disk Parameter header for disk 00
dpbase: defw  0000h, 0000h
        defw  0000h, 0000h
        defw  dirbf, dpblk
        defw  chk00, all00
; disk parameter header for disk 01
        defw   0000h, 0000h
        defw   0000h, 0000h
        defw   dirbf, dpblk
        defw   chk01, all01
; disk parameter header for disk 02
        defw   0000h, 0000h
        defw   0000h, 0000h
        defw   dirbf, dpblk
        defw   chk02, all02
; disk parameter header for disk 03
        defw   0000h, 0000h
        defw   0000h, 0000h
        defw   dirbf, dpblk
        defw   chk03, all03
;
; sector translate vector
trans: defm  1, 7, 13, 19    ;sectors 1, 2, 3, 4
       defm  25, 5, 11, 17   ;sectors 5, 6, 7, 6
       defm  23, 3, 9, 15    ;sectors 9, 10, 11, 12
       defm  21, 2, 8, 14    ;sectors 13, 14, 15, 16
       defm  20, 26, 6, 12   ;sectors 17, 18, 19, 20
       defm  18, 24, 4, 10   ;sectors 21, 22, 23, 24
       defm  16, 22          ;sectors 25, 26
;
dpblk: ;disk parameter block for all disks.
        defw   26    ;sectors per track
        defm   3     ;block shift factor
        defm   7     ;block mask
        defm   0     ;null mask
        defw   242   ;disk size-1
        defw   63    ;directory max
        defm   192   ;alloc 0
        defm   0     ;alloc 1
        defw   0     ;check size
        defw   2     ;track offset
;
; end of fixed tables
;
```

The key table, labeled dpblk, is the description of the floppy disk. The block shift factor, block mask, and null mask are linked to the block size, here 1024 bytes. See the Alteration Guide for details. The disk size tells CP/M that the disk has 243 blocks available for data. The directory max value tells CP/M to use at most 64 directory entries for each disk. The allocation values are a bitmap that shows CP/M which disk areas have the directories. Check size is a value used with changeable disks, set to zero here because our system has a hard disk. This is the only value in the dpblk table that I changed. (Note: I could have left it unchanged though). The track offset tells CP/M to reserve the first two tracks of each disk for system files.

SECTRAN and the Big Bug

I suppose with any good-sized software project you end up with a bug that makes you want to pull your hair out and/or drives you to prayer. Here is the one that got me.

You will notice the sector translate table in the code above. This is used by the BIOS SECTRAN subroutine to translate a logical sector request into a physical sector selection that is different from the logical sector. This was used by systems with floppy disks to increase efficiency, because the disk was much slower than the system, and waiting for a full disk rotation to read the next sector in physical order took a long time. So, the logical sectors were translated to physical sectors that were 6 sectors apart, all around the track. A simple system, easy to understand.

The translation table is connected to the rest of the CP/M by the SECTRAN subroutine in the BIOS. A SELDSK call returns the address of the disk parameter header for a particular disk, and the first item in that header is a pointer to the translation table. (The other items in the disk parameter header point to buffers and scratch areas for use by the CP/M BDOS). In the Alteration Guide, we are told that if we put 0x0000 in the place where the translation table pointer belongs, no translation will be done. Of course, in my system, no translation is needed, so I put 0x0000 there.

After this, I assembled my BIOS and tested it by placing it into memory using the ROM monitor bload command, and running short programs that used calls to the various BIOS routines for console input and output, and disk reads and writes. All worked perfectly. I prepared (formatted) the disk. I then created the CP/M system (essentially placed it into the first two sectors of the disk, as described in detail below). It worked -- sort of. CP/M would load and seem to start normally. But, directory listings were strange, sometimes one file was listed twice. Occasional disk read errors popped up. No amount of fiddling with the disk parameter table helped. I looked at the BDOS. Maybe some 8080 to Z80 translation error there. I looked at the CCP. Tried a few things. Hair pulling and prayer was done.

At the Lord's prompting I took a closer look at the SECTRAN subroutine. I had not changed it, because the Alteration Guide said if I put zero in the disk parameter header, no translation would be done. I was lied to.

Here is the original SECTRAN subroutine, fixed up after translation from the 8080 code by the toZ80 awk script. By fixed up I mean that the LD l,(hl) instruction originally came out as "LD l,m" -- the m should have been changed to (hl):

```
sectran:
        ;translate the sector given by bc using the
        ;translate table given by de
    EX      DE,HL  ;hl=.trans
```

```
ADD  HL,BC  ;hl=.trans (sector)
LD   l, (hl)  ;l=trans (sector)
LD   h, 0   ;hl=trans (sector)
ret         ;with value in hl
```

Note that there is no test to see if the pointer to the translation table (.trans) is zero. As such, if this subroutine is called, it will return a sector value that will be some random byte from the zero page of memory. I simply added a return after the exchange and addition of the logical sector and translation pointer (which will be zero), so it just returns the sector number it was called with:

```
sectran:
    ;translate the sector given by bc using the
    ;translate table given by de
EX   DE,HL  ;hl=.trans
ADD  HL,BC  ;hl=.trans (sector)
ret         ;debug no translation
LD   l, (hl)  ;l=trans (sector)
LD   h, 0   ;hl=trans (sector)
ret         ;with value in hl
```

After this CP/M ran perfectly. Apparently, some routines in the BDOS were calling SECTRAN without checking to see if translation was on or off.

Note that if I had not changed the first word in the disk parameter table header to zero all would have worked perfectly. That is, translation would have been done, but this would not be noticeable to the user. So, after everything, I learned that I did not need to make any changes at all in the disk parameter headers, or block tables, or the translate table to get CP/M to work properly on my system.

Preparing the Disk

This version of CP/M is meant to be installed into the first two tracks of the disk -- specifically track 0, sectors 2 though 26, and track 1, sectors 1 to 25. This is the area of the disk that the BIOS boot subroutines load into memory when CP/M is started or rebooted. I am referring to tracks on the the logical floppy disk described in the disk parameter tables -- remember we are using LBA to physically access the hard drive, translating the CP/M logical disk-track-sector numbers to a hard disk physical sector in the BIOS disk read and write subroutines. But, since the BIOS read and write subroutines are working perfectly when called after the appropriate SETDSK, SETDMA, SELTRK, and SELSEC calls, we will use these routines to access the disk whenever we can.

But, before installing CP/M, we need to prepare the disk for the CP/M file system. This is an easy task, since the only thing CP/M needs to see are empty directories in the disk areas where the directories will be found, as described by the disk parameter tables. The CP/M directory entry is a simple structure. It is 32 bytes long. The very first byte is the status of the directory. An empty directory has a status byte of 0E5h. If CP/M sees this byte, it considers the directory entry available, and will create an entry there. The remainder of the directory entry structure does not have to be created ahead of time, we only need to put OE5h in the first byte of the 64 directory

entries on each disk. In fact, it is easier than that -- we can just fill the whole disk with the byte 0E5h.

To do this, I wrote a simple "format" program that calls the BIOS subroutines to fill each CP/M disk with 0E5h. I used the RAM monitor bload command to load the assembled BIOS into memory at 0xFA00, then loaded the format program at 0x0300, and ran it. Here is the format program:

```
;Formats four classical CP/M disks
;Writes E5h to 26 sectors on tracks 2 to 77 of each disk.
;Uses calls to BIOS, in memory at FA00h
seldsk:         equ 0fa1bh              ;pass disk no. in c
setdma:         equ 0fa24h              ;pass address in bc
settrk:         equ 0fa1eh              ;pass track in reg C
setsec:         equ 0fa21h              ;pass sector in reg c
write:          equ 0fa2ah              ;write one CP/M sector to disk
monitor_start:  equ 0dc00h
                org 0300h
                ld  sp,format_stack
                ld  a,00h               ;starting disk
                ld  (disk),a
disk_loop:      ld  c,a                 ;CP/M disk a
                call seldsk
                ld  a,2                 ;starting track (offset = 2)
                ld  (track),a
track_loop:     ld  a,0                 ;starting sector
                ld  (sector),a
                ld  hl,directory_sector ;address of data to write
                ld  (address),hl
                ld  a,(track)
                ld  c,a                 ;CP/M track
                call settrk
sector_loop:    ld  a,(sector)
                ld  c,a                 ;CP/M sector
                call setsec
                ld  bc,(address)        ;memory location
                call setdma
                call write
                ld  a,(sector)
                cp  26
                jp  z,next_track
                inc a
                ld  (sector),a
                jp  sector_loop
next_track:     ld  a,(track)
                cp  77
                jp  z,next_disk
```

```
                inc  a
                ld   (track),a
                jp   track_loop
next_disk:      ld   a,(disk)
                inc  a
                cp   4
                jp   z,done
                ld   (disk),a
                jp   disk_loop
done:           jp   monitor_start
disk:           db   00h
sector:         db   00h
track:          db   00h
address:        dw   0000h
directory_sector:
                ds   128,0e5h      ;sector filled with 0E5h
                ds   32            ;stack space
format_stack:
                end
```

Putting CP/M on the Disk

Once I had assembled the CCP, BDOS, and customized BIOS programs, the next thing was to put them onto the disk. For this I wrote a putsys program. It uses calls to BIOS subroutines to copy a memory image of CP/M onto tracks 0 and 1 of the CP/M disk. Note track 0 sector 1 is reserved for boot code, not needed here.

This version of the putsys program was loaded and run from the ROM monitor.

```
;Copies the memory image of CP/M loaded at E400h onto tracks 0 and 1 of the first CP/M disk
;Load and run from ROM monitor
;Uses calls to BIOS, in memory at FA00h
;Writes track 0, sectors 2 to 26, then track 1, sectors 1 to 25
seldsk:             equ 0fa1bh  ;pass disk no. in c
setdma:             equ 0fa24h  ;pass address in bc
settrk:             equ 0fa1eh  ;pass track in reg C
setsec:             equ 0fa21h  ;pass sector in reg c
write:              equ 0fa2ah  ;write one CP/M sector to disk
monitor_warm_start: equ 046Fh   ;Return to ROM monitor
                    org 0800h   ;First byte in RAM when memory in configuration 0
                    ld  c,00h   ;CP/M disk a
                    call seldsk
;Write track 0, sectors 2 to 26
                ld   a,2            ;starting sector
                ld   (sector),a
                ld   hl,0E400h      ;start of CCP
```

```
                ld   (address),hl
                ld   c,0            ;CP/M track
                call settrk
wr_trk_0_loop:  ld   a,(sector)
                ld   c,a            ;CP/M sector
                call setsec
                ld   bc,(address)   ;memory location
                call setdma
                call write
                ld   a,(sector)
                cp   26             ;done:
                jp   z,wr_trk_1     ;yes, start writing track 1
                inc  a              ;no, next sector
                ld   (sector),a
                ld   hl,(address)
                ld   de,128
                add  hl,de
                ld   (address),hl
                jp   wr_trk_0_loop
;Write track 1, sectors 1 to 25
wr_trk_1:       ld   c,1
                call settrk
                ld   hl,(address)
                ld   de,128
                add  hl,de
                ld   (address),hl
                ld   a,1
                ld   (sector),a
wr_trk_1_loop:  ld   a,(sector)
                ld   c,a                ;CP/M sector
                call setsec
                ld   bc,(address)       ;memory location
                call setdma
                call write
                ld   a,(sector)
                cp   25
                jp   z,done
                inc  a
                ld   (sector),a
                ld   hl,(address)
                ld   de,128
                add  hl,de
                ld   (address),hl
                jp   wr_trk_1_loop
done:           jp   monitor_warm_start
```

```
sector:     db   00h
address:    dw   0000h
            end
```

The file cpm.sys which contains the assembled code for the CCP and BDOS was first loaded into memory at location 0xE400. Then, the BIOS was loaded at location 0xFA00. Then the putsys program was loaded at location 0x0800 and run. The drive activity light went on, and then off, and CP/M was now on the disk.

The CP/M Loader

To get CP/M from the disk into the system memory at computer startup, I created a cpm_loader program. Note that CP/M uses disk sectors 1 to 26 of each track. That means that the hard disk LBA sector 0 was available for use. Once I assembled the cpm_loader, I put the machine code into LBA sector zero using the ROM monitor diskwr command.

The ROM monitor, which starts when the computer is turned on, or taken out of reset, has a command cpm that loads the full 256 bytes of hard disk sector 0 into RAM at location 0x0800 and jumps to it. Here is the ROM monitor code:

```
cpm_jump:  ld   hl,0800h
           ld   bc,0000h
           ld   e,00h
           call disk_read
           jp   0800h
```

It uses the ROM monitor disk_read subroutine that takes BC and E as the LBA, and HL as the memory area to write to. Here is the CP/M loader code:

```
;Retrieves CP/M from disk and loads it in memory starting at E400h
;Uses calls to ROM subroutine for disk read.
;Reads track 0, sectors 2 to 26, then track 1, sectors 1 to 25
;This program is loaded into LBA sector 0 of disk, read to loc. 0800h by ROM
disk_read subroutine, and executed.
;
hstbuf:     equ 0900h   ;will put 256-byte raw sector here
disk_read:  equ 0294h   ;subroutine in 2K ROM
cpm:        equ 0FA00h  ;CP/M cold start entry in BIOS
            org 0800h   ;Start of RAM, configuration 0
;Read track 0, sectors 2 to 26
                ld   a,2            ;starting sector -- sector 1 reserved
                ld   (sector),a
                ld   hl,0E400h      ;memory address -- start of CCP
                ld   (dmaad),hl
                ld   a,0            ;CP/M track
                ld   (track),a
rd_trk_0_loop:  call read
                ld   a,(sector)
```

```
                cp   26
                jp   z,rd_trk_1
                inc  a
                ld   (sector),a
                ld   hl,(dmaad)
                ld   de,128
                add  hl,de
                ld   (dmaad),hl
                jp   rd_trk_0_loop
;Read track 1, sectors 1 to 25
rd_trk_1:       ld   a,1
                ld   (track),a
                ld   hl,(dmaad)
                ld   de,128
                add  hl,de
                ld   (dmaad),hl
                ld   a,1            ;starting sector
                ld   (sector),a
rd_trk_1_loop:  call read
                ld   a,(sector)
                cp   25
                jp   z,done
                inc  a
                ld   (sector),a
                ld   hl,(dmaad)
                ld   de,128
                add  hl,de
                ld   (dmaad),hl
                jp   rd_trk_1_loop
done:           out  (1),a          ;switch memory config to all-RAM
                jp   cpm            ;to BIOS cold start entry
read:
;Read one CP/M sector from disk 0
;Track number in 'track'
;Sector number in 'sector'
;Dma address (location in memory to place the CP/M sector) in 'dmaad' (0-65535)
;
                ld   hl,hstbuf      ;buffer to place raw disk sector (256 bytes)
                ld   a,(sector)
                ld   c,a            ;LBA bits 0 to 7
                ld   a,(track)
                ld   b,a            ;LBA bits 8 to 15
                ld   e,00h          ;LBA bits 16 to 23
                call disk_read      ;subroutine in ROM
;Transfer top 128-bytes out of buffer to memory
```

```
                ld      hl,(dmaad)      ;memory location to place data read from disk
                ld      de,hstbuf       ;host buffer
                ld      b,128           ;size of CP/M sector
rd_sector_loop: ld      a,(de)          ;get byte from host buffer
                ld      (hl),a          ;put in memory
                inc     hl
                inc     de
                djnz    rd_sector_loop  ;put 128 bytes into memory
                in      a,(0fh)         ;get status
                and     01h             ;error bit
                ret
sector:         db      00h
track:          db      00h
dmaad:          dw      0000h
                end
```

The assembled code for this program takes up only 136 bytes so it fits easily into the single 256-byte sector alloted for it. I could also have put it into CP/M track 0 sector 1.

Loading Programs into CP/M through the Serial Interface

With CP/M loaded onto disk, to start it, one simply issues the cpm command at the ROM monitor prompt. The CP/M loader puts CP/M into its proper place in memory, changes the memory configuration to all-RAM, and jumps to the BIOS cold start entry point to start CP/M. The CP/M prompt A> appears, and we can enter commands.

CP/M has a small number of internal, or built-in commands -- DIR to list the directory of a disk to the screen, ERA to erase a file, and a few others. See the System Manual for details. But, there is no built-in command available to check disk space, copy files, edit text, assemble code, or do most other things that we would expect an operating system to do. There is no command to load a file into the disk through the serial port. Once CP/M is working properly, we need a way to get programs into the CP/M disk file system.

Fortunately, there is a built-in command SAVE that will write code from memory into a CP/M file. The SAVE command will take a number of 256-byte pages from memory, starting at location 0x0100, and create a file. So, our task is to use the RAM monitor program bload command to load a CP/M program file through the serial port into memory at 0x0100, then use the CP/M SAVE command to create a program file. Once that file is created, to run the program, one types the program name at the CP/M prompt. CP/M loads the file into memory at location 0x0100, and jumps to that location to execute it.

Since we only have one serial port, and CP/M dedicates that port to console input and output, we have to work around CP/M to do the binary loads through that port. To get the process started, we need to have the both the RAM monitor program and CP/M present in memory at the same time. To do this, we start by loading the RAM monitor at location 0xDC00, using the ROM monitor bload command. Once the RAM monitor is in the memory, we load CP/M into memory using the ROM monitor cpm command. Now, both programs are in memory at the same time, and we are running CP/M. But, we need to be in the RAM monitor to bload programs into memory. There

is no CP/M command to jump to a particular memory location. How to get from CP/M to the RAM monitor?

Simple. We reset the computer. A computer reset does nothing to the RAM system memory -- it leaves the memory contents unchanged. If we reset the computer, then take it out of reset, we will be in the ROM monitor, with the memory in configuration 0. Then we can use the run command to jump to the RAM monitor which is in memory at 0xDC00. The first instruction in the RAM monitor changes the memory configuration to 1, so now we have access to the lower RAM locations and can bload a program into memory at 0x0100 using the RAM monitor command line. Then, we go to CP/M by using the monitor run command with the address 0xFA00 (BIOS cold start entry). (Note that even though the RAM monitor, like the ROM monitor, has a cpm command, we must not use this to start CP/M, because it will load the cpm loader program into memory at 0x0800, which might overwrite the program we previously loaded, the one we want to save.) Jumping to location 0xFA00 in the BIOS re-loads CP/M into memory, but will not touch the program file previously loaded at 0x0100. Now, at the CP/M prompt we can issue the SAVE command, and the memory contents will be saved as a file on the CP/M disk and placed in the directory, ready to be used.

Which program should we start with? The first should be the MONITOR.COM program. This is a version of the RAM monitor designed to be called and used by CP/M. MONITOR.COM is 2008 bytes long. 2008 divided by 256 is 7.84, so we need to save at least 8 pages of memory. After bloading it to location 0x0100 with the RAM monitor, we issue this command:

A>SAVE 8 MONITOR.COM

A DIR command will show the new file in the disk directory. Now, if we want to load a file through the port, we can simply issue the CP/M command MONITOR and we will get the monitor prompt. The MONITOR program, when executed, moves itself from location 0x0100 where CP/M places it, to location 0xDC00. It uses buffer space, variables, and stack space in memory page 0xDB00 also. This allows us to use the memory space from 0x0100 to 0xDAFF, or 55,807 bytes, to load programs for saving in the CP/M file system. And, once a file is loaded into memory using the MONITOR program, we can return to CP/M by issuing the cpm command at the MONITOR program prompt -- this command does not affect the low program memory, just jumps to the CP/M BIOS cold entry start point, without using the cpm loader program, so the memory is left undisturbed. Then we can issue a CP/M SAVE command.

Once the MONITOR command is working, we can load the rest of the important CP/M command files. There is STAT, which shows statistics for file sizes and disk room available. There is the ED command, which is a text editor. I got a smile out of using it, it is like editing a ghost, because you cannot see the text you are editing -- you are moving an invisible cursor through an invisible text buffer in memory to make changes, which you can see afterward using the T (type) command. There is the PIP command, for file copying. There are the ASM and LOAD commands for assembling 8080 files. There are also Z80 assemblers. Finally, there are hundreds of CP/M programs available on-line that you can download and try. The Humongous CP/M Archive is a good starting point. I found the chess program Sargon, and played a game against my computer. It won -- maybe I let it win, but I don't think so.

Future Tasks

Now that I have a working CP/M 2.2 system, I may try to create a better BIOS. Perhaps I will have it print out a greeting message at cold boot. I would probably remove

unnecessary code, like the unused translation table. I might work on the disk read and write routines to make more efficient use of the disk. In the current configuration, there are 4 disks (2 bits) of 77 tracks (7 bits) with 26 sectors (5 bits) on each track. That means 14 bits is enough to give each sector in the whole CP/M disk system a unique LBA by a simple OR operation. This would be important if one wanted to use a small disk, like a 16 MB compact flash drive. I might try different disk parameters in the tables. I need to create some easy way to get files out of CP/M, probably by writing a CP/M program that loads a file into memory without executing it, and then using the MONITOR to bdump the memory containing the file to the serial port. I might try to get CP/M 3 up and running. And I will certainly play chess with my computer again.

--Donn Stewart, July 2014
