Workshop Series

Bringing up CP/M on Your S-100 Floppy Drive System

Richard Cini – Altair32, N8VEM

Agenda and Outline

- Introduction and my story
- Understanding the structure of CP/M
- Tools and items required
  - What you should have on-hand
- The process itself
  - How to bring-up a new system
- Q&A

Introduction

My IMSAI as an Example

- I “adopted” my system from a gentleman in Arizona who was downsizing.
- This system had a partially-working iCOM “Frugal Floppy” 8” drive system (now part of the MARCH collection). Drives were unreliable and disks that worked contained CP/M 1.4. Controller is proprietary TTL.
- Memory size was 48k of RAM in three, 16k boards. Uncommon SRAM chip used.
- Serial console card – working (proven through booting existing CP/M).
- No obvious way to regenerate system disks or move to CP/M 2.2. Can’t use cross-platform image tools because FD400 floppy can’t be connected to a PC controller.

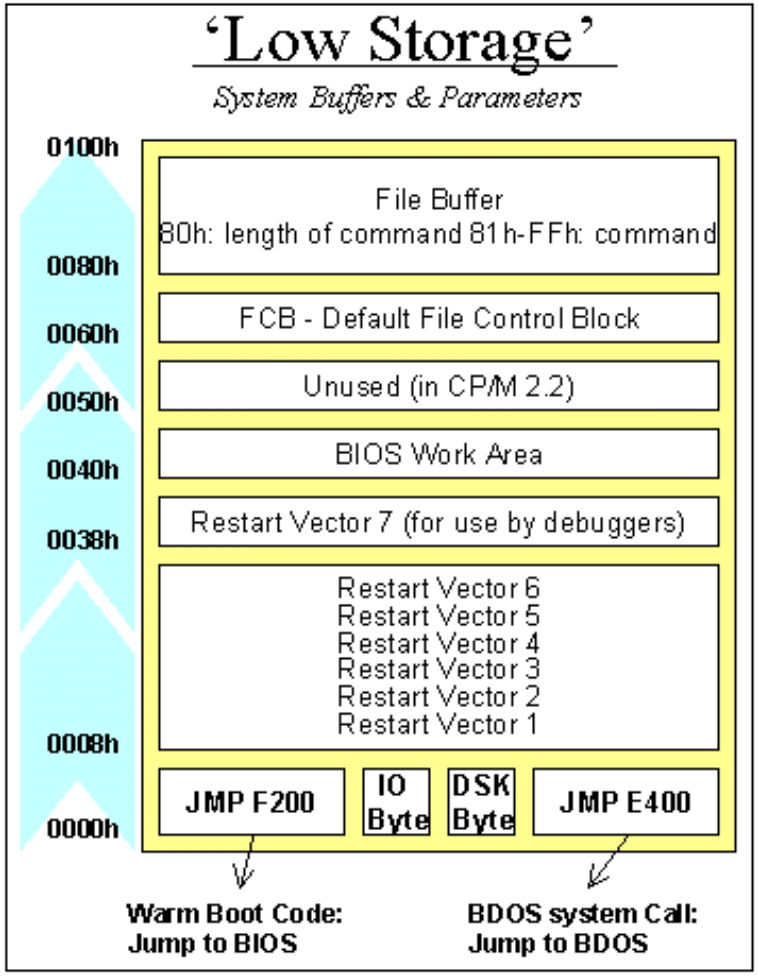
My IMSAI (con't)

- Hardware Check-out: serial card and memory checked-out.
- Alternate ROM monitor:
  - There was simple ROM monitor on the iCOM disk controller card, but it had no HEX loader.
  - Found a good basic monitor program from Dave Dunfield. Made changes for the SSM card and burned to 2716 for use on an EPROM card. PASS
- This is the stage at which the system sat for months because I couldn't get the iCOM system working consistently, and only working drive was beginning to work only intermittently.
- Finally shelved the iCOM system and started locating 8"/5.25" intelligent floppy controller cards to build a setup from scratch.
- Help arrives from Internet friend with same system.

Understanding CP/M

CP/M Internals

- A CP/M system memory map is comprised of several regions (from lowest address to highest):
  - "Low Storage" contains system jump vector, restart vectors, default FCB and file buffer area.
  - TPA: Transient Program Area (the "user" area where programs are loaded).
  - CCP: Console Command Processor. This is the start of CP/M itself.
  - BDOS: Basic Disk Operating System. Contains callable DOS APIs. BDOS calls routines in the CBIOS through standardized jump table located between BDOS and CBIOS.
  - CBIOS: User-supplied Customized Basic I/O System. Accessed through the jump table in BDOS. This functions as a hardware abstraction layer.



Source: Oscar Vermeulen

CP/M Internals (con't)

- "CP/M" is defined as the CCP+BDOS (supplied by DRI) plus CBIOS (user-supplied). All parts of CP/M (through the CBIOS jump table) have to be in contiguous RAM memory.
- Each part of CP/M is located at a fixed address based on memory size.
- Minimum memory requirement is 20K, resulting in a TPA of about 13K.
- Memory size was based on need…RAM was very expensive back then. Could be expanded easily as needs changed.
- Booting: first-level bootstrap (GETSYS) in EPROM loads second-level cold-start loader from track 0/sector 1 of disk; remaining CP/M image loaded by cold-start loader.

Obtaining CP/M

- Purchase “integrated” system from local computer store or semi-integrated from multiple mail-order vendors.
- There was no “standard” system configuration other than assuming you had RAM, a console card and a disk controller. Most manuals came with sample code for modifications.
- Disk controller or system usually came with manual and CP/M disk that had to be customized for specific hardware configuration unless vendor did it as part of their sale value-add.
- CP/M can’t auto-configure for different hardware.
- Rely on local help (store gurus, users groups) to assist with building CBIOS from template.

CP/M the Modern Way

- Decompiled CP/M source code available publicly.
- Still need to build customized BIOS for your system.
- Ability to use either legacy or modern tool chains, including emulators.
- Flexibility as to how CP/M loads (all from disk or hybrid disk/ROM)
- May use one method to produce first CP/M disk and another method later.
- Sample source code more readily available in on-line legacy archives like CPMUG or SimTel.

The Modern Way (con't)

- More flexible memory arrangements potentially allow for packing memory tighter, resulting in larger TPA.
- Ability to use external drive emulators like the Altair Peripheral Emulator (http://home.comcast.net/~forbin376/) for bootstrapping.
- Phone a friend.

The Rebuild Process

First Steps

- Basic troubleshooting and system evaluation/qualification:
  - Inventory hardware; locate manuals for every board.
  - Basic testing, repair and validation.
  - Enough working RAM cards to get 32k+?
  - Need EPROM board? Maybe switch RAM to card which can support EPROMs, like the CompuPro RAM17 (6116 == 2716). I eventually did this to fill-out memory and use EPROM card in other system.
  - Working serial console card?
- Is there an existing ROM monitor? Can it load Intel HEX files?
  - This is key because HEX loader is eventually used to get CP/M into memory for the first time. Get simple ROM monitor working before tackling disk system.

First Steps (con't)

- Begin to evaluate disk system:
  - Is existing disk system in reasonable condition? Proprietary or standard controller and interface?
  - Stick with 8" (Shugart, Qume, others) or use 5.25" HD (YD-380) as equivalent to replace failing 8" drives.
  - Convert to native 5.25"/360k? Larger, like 3.5"/720k (hard to find as native), or 3.5"/1.44mb.
  - If using PC-compatible drives, check them out on a PC first.
  - Check jumpers and cabling.
  - Can you communicate with controller card? For intelligent controllers, the "seek" command is easiest to use (3-byte controller command: $0F, $0, $0 to seek track 0).
  - Again, basic blocking-and-tackling to ensure you can communicate with controller card.

First Steps (con't)

- Some controller cards configurable for 8" or 5.25" drives, but there are some gotchas when using modern drives with old controllers
- Modern drives have signals in different places and controllers have signals that are no longer supported by modern drives:
  - Floppy *READY signal on pin 34 (new drive: disk_change)
  - Head_load (HL) on pin 2 (new drive: density_select)
  - Drive outputs *READY on pin 4 (controller pin 34).
- May need to re-jumper or modify drive to work with controller. Follow the manual and schematic.

CP/M Load Methods

- Contiguous Disk Load:
  - CP/M (CCP+BDOS+CBIOS) loaded from disk by small bootstrap program in EPROM.
  - CBIOS changes require regenning CP/M and copying to every bootable disk.
  - Still has its place today if bootable disks already exist.
  - Good if using system for programs other than CP/M (fewer ROM holes).
- Split Disk Load/ROM:
  - Changes to CBIOS that don't impact the jump table won't require re-copying updated system to every boot disk. If there's room, you can build "patch space" into the ROM to keep addresses aligned.
  - Can fragment memory depending on flexibility of EPROM board, but could also increase TPA through address space compaction.
  - PUTCPM routine in ROM makes it easy to create new bootable disks.
  - Good choice for modern rebuilds.
- ROM CP/M:
  - Used in some SBC systems like the N8VEM. Moved to RAM from banked ROM by reset bootstrap.

Building a Memory Map

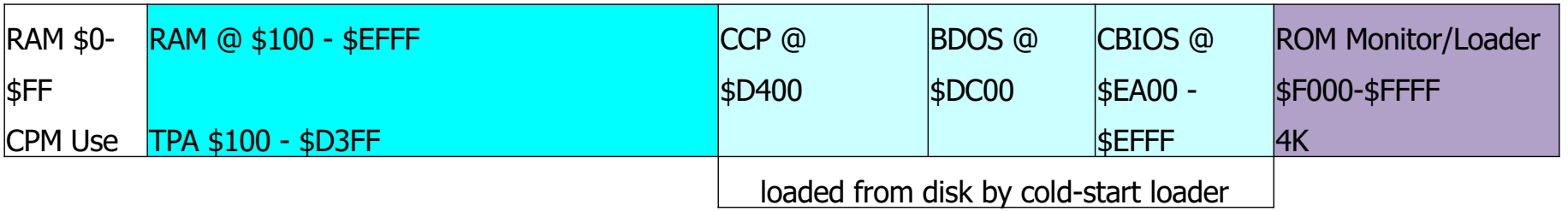
- CP/M requires about 5.5k for code (not including CBIOS) and RAM area for working variables and buffers.
- CBIOS code "budget" is 1.5k. My CBIOS requires about 1.9k for code, but it includes ROM monitor, disk formatter, and APE bootstrap.
- RAM usage includes 1k for track buffers and 350 bytes for variables.
- How flexible and configurable is memory system?
  - One memory board or multiple boards? Board density?
  - Separate EPROM board?
- Start thinking about location of and method to load CP/M:
  - Depends on state of repair. If you have nothing, it might be easier to split CCP+BDOS from CBIOS (split loadable).

Building a Memory Map (con't)

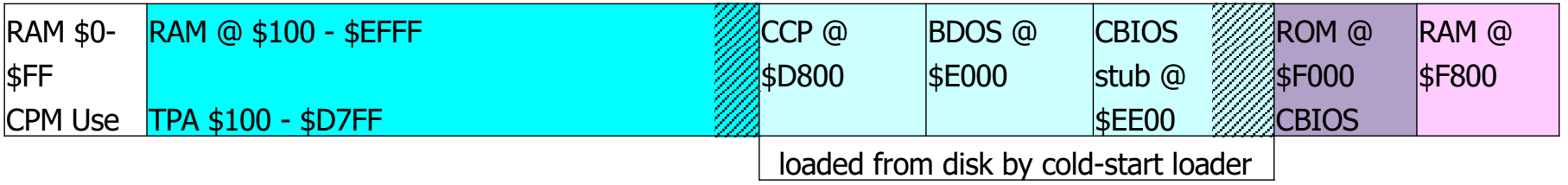
- Since I was basing my configuration on someone else's system, I needed to have my system look as much like that system as possible. ROM monitor also contained functional CBIOS.
- I used a CompuPro RAM17 (6116 SRAM chips); 2716 drop-in at $F000; disk buffers and working variables at $F800. Avoids having to use separate EPROM card (I had only one).
- In this configuration, CP/M "size" won't necessarily match actual contiguous RAM memory size because of ROM boundaries in comparison to CP/M Alteration Guide address table.
  - Use right system "size" number to get addresses in the right place.
  - CP/M configured as a "61K" system but only has 60K contiguous.
  - 61K size picked as it places CBIOS stub ending address closest to (but not over) the starting address of ROM monitor.
  - Some small amount of wasted address space between CBIOS stub and actual CBIOS code in ROM, but that's the price you pay.
  - Putting buffers/variables above EPROM added to TPA.

Building a Memory Map (con't)

Traditional 60K CP/M Memory Layout



Layout for my 61K IMSAI System



Building a Memory Map (con't)

Equates area of CBIOS shows how to calculate the CP/M addresses. "20" is the minimum memory size; the calculated BIAS amount (address offset) enables relocating the image for different memory sizes.

Table on next page contains pre-calculated addresses.

```
;:::::::::::::::::::::::::::::::::::::::::::::::
;   S Y S T E M   E Q U A T E S
;*********************|
MSIZE       .EQU 61 ;|  ; CP/M system size in K
;*********************|
; CP/M Definitions - Valid for version 2.2 only
K           .EQU        1024
BIAS        .EQU        (MSIZE-20)*K
CCP         .EQU        3400h+BIAS  ; CCP ENTRY POINT
BDOS        .EQU        CCP+800h+6  ; BDOS ENTRY POINT
CBIOS       .EQU        CCP+1600h   ; CBIOS jump table
```

Memory Map (con't)

BIOS Length	BIOS Base	DDT Offset	MOVCPM 'nn'	CCP Base	BDOS Base
600	FA00	2580	64	E400	EC00
A00	F600	2980	63	E000	E800
E00	F200	2D80	62	DC00	E400
1200	EE00	3180	61	D800	E000
1600	EA00	3580	60	D400	DC00
1A00	E600	3980	59	D000	D800
1E00	E200	3D80	58	CC00	D400
2200	DE00	4180	57	C800	D000
2600	DA00	4580	56	C400	CC00
2A00	D600	4980	55	C000	C800
2E00	D200	4D80	54	BC00	C400
3200	CE00	5180	53	B800	C000
3600	CA00	5580	52	B400	BC00
3A00	C600	5980	51	B000	B800
3E00	C200	5D80	50	AC00	B400
4200	BE00	6180	49	A800	B000
4600	BA00	6580	48	A400	AC00
4A00	B600	6980	47	A000	A800
4E00	B200	6D80	46	9C00	A400
5200	AE00	7180	45	9800	A000
5600	AA00	7580	44	9400	9C00
5A00	A600	7980	43	9000	9800
5E00	A200	7D80	42	8C00	9400
6200	9E00	8180	41	8800	9000
6600	9A00	8580	40	8400	8C00
6A00	9600	8980	39	8000	8800

Source: Programmer's CP/M Handbook

Building CP/M the DRI Way

- CP/M Alteration Guide written from the perspective of having access to new system master disks or other way of obtaining original disks.
- Two-step process:
  - Configure distribution disk for correct hardware at 20K default memory size (“first level system generation”)
  - Regenerate system for actual RAM size (“second level system generation”). My IMSAI would be a “61K System”.
  - System sizes were usually integral multiples of 4K.

Building CP/M the DRI Way (First Level)

- Write and test GETCPM and PUTCPM routines. These routines read and write the system tracks.
- Test on uninitialized disk.
- Write and test CBIOS code; start with simple routines and work up: console, disk
- All patching occurs in memory and written to disk using PUTCPM routine.
- Write bootstrap for track 0/sector 1 based on working GETCPM code above. Write it to disk. This code can also be in EPROM.
- Always work with scratch diskette until certain everything works.
- Results in working CP/M system but at a 20K memory size.

Building CP/M the DRI Way (Second Level)

- Second level system generation re-sizes working 20K system for the actual memory size.
- Again, another involved process:
  - Make any required changes to customized CBIOS and CBOOT using tools on 20K system (ED and ASM). Recompile and save as HEX files.
  - Load CP/M as a relocated image into TPA from system tracks using MOVCPM with parameter for new memory size.
  - Use SAVE command to save relocated system to a file. Code is located relative to start of TPA (100h) rather than where in memory it should be.
  - Use DDT commands to adjust addresses so that they're in the right place for new memory size.
  - Use SYSGEN to write relocated system to new blank diskette.

Items Required to Build Initial Disk using Today's Tools

- Of course, tested and working hardware.
- Source code for CP/M (CCP+BDOS). Grab source from Gaby.de. Only the "memory size" and load address parameters get changed based on address table in Alteration Guide.
- Skeletal CBIOS code from DRI manual or elsewhere. This is what gets modified to match your hardware. To save some work, try to locate one for a similar disk controller card.
- EPROM-related tools if needed.
- Software tools: cross-compiler or native compiler using available CP/M emulations (MyZ80 or SIMH).
- Related tools: binary file editor, binary-to-hex converter, text editor.
- CP/M books and user's group archives.

Building CP/M Using Split Loading Method

- This is the method I used on my IMSAI.
- Method relies on the fact that part of CP/M (CCP+BDOS) resides on the floppy disk and part (CBIOS) resides in EPROM.
- CCP+BDOS compiled separately; minimal development work needed other than setting the right memory size and compiling. Object code merged with the CBIOS stub and written to disk at the very end of the process.
- This leaves development time free for working with the CBIOS in EPROM using more modern tools on a different platform (Mac/PC).
- Might not always be the best way or final way for getting a system running. After bootstrapping this way, one might want to try building a new master disk exactly like an original DRI disk (CCP+BDOS+CBIOS all loaded from disk).
- APE Disk Emulator is an option as well.

Building CP/M Using Split Loading Method (CP/M)

- Used memory map to determine that I needed to configure CP/M for a 61K system. Change two equates at top of cpm22 source file.
- With split-loaded CP/M, some trickery is needed since CP/M and CBIOS reside in different source files and because of EPROM boundaries, addresses used won't be perfectly linear.
- CCP+BDOS is compiled with "fake" CBIOS jump table at the expected address so that target addresses in the source file are valid (but point to 0).
- CCP+CBIOS code from gaby.de uses CP/M ASM directives so rather than make changes for other tools, I used ASM under MyZ80 to compile code (need to first import asm.com to disk image):
  - Import source file: import cpm22.asm
  - Compile: asm cpm22.asm
  - Export: export cpm22.hex
- Need to patch the cpm22.hex file later with jump table output from CBIOS compilation (HEX file is simple text file; easy to patch).

Building CP/M Using Split Loading Method (CBIOS)

- CBIOS is similar to IO.SYS in MS-DOS
- First block of code is a function jump table; called by the BDOS (similar to MSDOS.SYS).
- Core CBIOS can be combined with other “useful” routines for disk formatting and/or system monitor (common with EPROM).
- NO inline code (i.e., no RETs) allowed in jump table; must only be JMP instructions followed by target address.
- Certain programs, like MBASIC and BYE, rely on structure and ordering of jump table in order to perform address intercepts and redirects.
- Routines grouped by general function: booting, console, list, punch, and floppy disk, and reflect hardware available at the time.
- Most hardware manuals come with sample code that can be used in modifying the “skeletal BIOS” from DRI or elsewhere.

Building the CBIOS
Primitives

```
CBios   .equ    0EE00h  ;Start of CBios-61k system
;
        .org    CBios
;
;:::::::::::::::::::::::::::::::::::::::::::::::::
;       J U M P S - jump table defs
;:::::::::::::::::::::::::::::::::::::::::::::::::
; sample jump table from CBIOS
JMP     CBOOT           ;Cold boot
JMP     WBOOT           ;Warm boot

JMP     CONST           ;Console status input
JMP     CONIN           ;Console input
JMP     CONOUT          ;Console output

JMP     LIST            ;List output
(oddly, list status is at the bottom)

JMP     PUNCH           ;Punch output
JMP     READER          ;Reader input

JMP     HOME            ;Set track to zero
JMP     SETDSK          ;Select disk unit
JMP     SETTRK          ;Set track
JMP     SETSEK          ;Set sector
JMP     SETDMA          ;Set Disk Memory Address
JMP     CREAD           ;CPM Read from disk
JMP     CWRITE          ;CPM Write to disk
JMP     LISTST          ;List status (output)
JMP     SEKTRN          ;Translate sector number
```

- CBOOT/WBOOT – cold-start and warm-start routines to load/reload CP/M into RAM.
- CONST/CONIN/CONOUT – console access routines for channel status, input and output.
- LIST/LISTST – printer device output and status routines.
- PUNCH/READER – device driver for paper tape punch/reader.
- Floppy disk – track/sector selection, read/write primitives. SEKTRN translates logical sector # (zero-based) to physical sector on disk (interleave table).

Building the CBIOS (con't)
Key Functions

- CBOOT (the cold-start loader):
  - Loads CP/M image from disk using GetCPM routine.
  - Sets initial variables and jumps to CCP.
  - EPROM may or may not have code to bring entire CP/M image into RAM (otherwise must use bootstrap program on disk).
- WBOOT (the warm-start loader):
  - re-loads CP/M from disk when user program exits by terminate call or when Control-C pressed to log in a new disk.
- Console in/out/status; list; punch/reader:
  - Code to talk to these devices is very simple.
- Disk code:
  - More complex; intelligent controllers use multi-byte controller commands.
  - Sector blocking/deblocking based on interleave; can ignore if disk is formatted with interleave.
  - Drive select logic.
  - Need to translate BDOS calling parameters to ones useful for controller. Lots of MOV/RAL/ROR.

Building the CBIOS (con't)
Preparing for Floppies

- Manuals usually have details on configuring for CP/M and include a template CBIOS to work with.
- Two key data items for floppies:
  - DPB (Disk Parameter Block):
    - 15-byte data block that describes disk geometry
    - One DPB per each different type of drive
  - DPH (Disk Parameter Header):
    - 16-byte data block storing logical drive info
    - One DPH for each available logical drive letter
    - Points to directory buffer and related DPB

Building the CBIOS (con't)
DPB and DPH

```
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;D P B ' S - DISK PARAMETER BLOCKS.
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;THE FORMAT OF THESE AREAS ARE AS FOLLOW:
;     DW = SECTORS PER TRACK.
;     DB = BLOCK SHIFT, BS MASK, EXTENT MASK.
;     DW = MAX ALLOCATION BLOCK NUMBER - 1.
;     DW = DIRECTORY ENTRIES NUMBER - 1.
;     DW = ALLOCATION BLOCKS BITMAP FOR DIRECTORY.
;     DW = CHECK AREA BUFFER SIZE (1BYTE/4 ENTRIES).
;     DW = TRACKS OFFSET BEFORE DIRECTORY (SYSTEM AREA).
;
;     Use one DPB per *type* of drive, not for each drive
;     if they are the same drive type.
;
DPBS2                         ;DSSD 8" DISK
      .DW      26             ;26 SECTORS / TRK
      .DB      4,15,1         ;2048 BLOCK SIZE PARAMETERS
;((77*2SIDES-2)*26SECS)/(2048/128 SEC/BLK)-1 AND ROUND DOWN
      .DW      245
      .DW      127            ;64 ENTRIES/STD 8" DISK SIDE-1
      ;2 ALLOC BLOCKS (128ENT X 32B/ENT / 2048B/BLK)
      .DB      11000000B,00000000B
      .DW      32             ;128ENT / 4ENT/BYTE
      .DW      2              ;2 TRKS BEFORE DIRECTORY
```

```
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;D P H ' S - DISK PARAMETER HEADERS
;COPY THESE -> RAM @DPHBASE BEFORE CPM RUN
;One for each block device
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
DPHROM
;     LOGICAL DISK A 8" FLOPPY
      .DW      0              ; 8" SKEW TABLE VECTOR
      .DW      0,0,0          ; CPM RESERVED
      .DW      DIRBUF         ; DIRECTORY BUFFER VECTOR
      .DW      DPBS2          ; DISK PARAMETER BLOCK VECTOR
      .DW      DSK1CV         ; CHECK VECTOR (was NO)
      .DW      DSK1AV         ; ALLOCATION VECTOR

;     LOGICAL DISK B 8" FLOPPY
      .DW      0              ; 8" SKEW TABLE VECTOR
      .DW      0,0,0          ; CPM RESERVED
      .DW      DIRBUF         ; DIRECTORY BUFFER VECTOR
      .DW      DPBS2          ; DISK PARAMETER BLOCK VECTOR
      .DW      DSK2CV         ; CHECK VECTOR (was NO)
      .DW      DSK2AV         ; ALLOCATION VECTOR
```

Building the CBIOS (con't)
Optional Functions

- Not part of a standard CBIOS:
  - System monitor
  - Blank disk formatting. Usually separate CP/M program but nice to have in-ROM.
  - PUTCPM to write CP/M to system tracks.
  - Non-standard hardware drivers like video or RTC

Building the CBIOS (con't)
Compiling & Merging

- Steps will vary based on tool used. I use TASM (Table Assembler) to assemble the CBIOS.
- Need to produce both binary and HEX file output which can be used to burn into ROM and to patch the jump table in the HEX file from compiling CCP+BDOS.
  - Since my EPROM programmer software isn't good, I had to compile twice to produce both HEX and a straight binary file for burning to EPROM. Others may be able to do this in one step.
  - Simple cut-and-paste job to remove null jump table from CCP+BDOS and insert active jump table from CBIOS object file.
  - Remaining CBIOS object code is burned to ROM.
- HEX files are plain text files which contain memory location information, so when they're loaded into memory using ROM monitor, the code ends-up at the right addresses in RAM.
- Once CCP+BDOS is in memory (with right jump table), use PUTCPM utility to write it to the system tracks on a blank disk.
  - Need to figure out number of sectors to write based on contiguous address space used.

Formatting A Disk

- Format utility writes $E5 to all data areas of disk.
- Walk tracks and translate to T/H/S for controller.
- Can be a separate program or part of an enhanced monitor EPROM.
- For intelligent controllers, build format command buffer; send to controller as a format command.
- Controller chip takes care of writing sync and track ID bytes.
- Other controllers may not be as programming friendly so refer to the manual.

Getting CP/M Onto Disk

- "PUTCPM" utility configured to write specific memory region to the system tracks.
- Can be a separate program or part of an enhanced monitor EPROM.
- My CBIOS relies on multi-sector write capability of 8272; other controllers may be different.
  - Need to specify starting DMA address and number of whole (128-byte) sectors to write, up to 26.
  - Repeat as necessary; my CP/M needed 45 sectors, which is two separate write commands.
  - Controller takes care of the heavy lifting.

Questions?

Historical Context

- Early systems had limited storage capabilities
  - paper tape or audio cassette
  - low storage density; sequential access
  - good for small programs or limited data
- Growth in adoption of platform demanded better storage options
  - floppy systems already existed on mini-computers
- Expandable S-100 buss and available 8” floppy drives served growing user base

Historical Context (con't)

- Initial floppy systems were expensive
  - $1,500 for single Altair 88-DCDD controller and FD400 drive. 237.25k formatted capacity.
- Increased demand for smaller form factor and lower cost. Push by Wang Labs for form factor smaller than 8" for new desktop word processing system being developed.
- Shugart Associates developed 5-1/4" SA400 (SSSD) and introduced it in 1976
  - Cost: $425.
  - Capacity: 89.6k formatted (110k unformatted)

Historical Context (con't)

- Two different sectoring types (VHS versus Beta)
  - Hard-sectored (90k unformatted)
  - Soft-sectored (110k unformatted)
- 5-1/4" quickly displaced 8" and hard-sectored format eventually disappeared
- By 1978, there were about 10 mini-floppy manufacturers.

Historical Context (con't)

- 1978 leap to double-sided recording and then double-density. 360k capacity.
- 1982 introduction of the YD380 high-density (1.2mb) format used on the PC/AT.
- The YD380 can substitute for an 8" drive in restoring a system as it runs at same RPM and transfer rate.