CUSTOMIZING MS-DOS version 1.23 and later

Setting the Special Editing Commands

The escape codes used by Function 10, buffered console input, can be set for the convenience of the user, using a table starting at address 0003 in MS-DOS. The beginning of MS-DOS looks like this:

```
0000            JMP     INIT

        ESCCHAR:
0003            DB      1BH     ;ASCII value to use for escape character

        ESCTAB:
0004            DB      "S"     ;Copy one character from template
0005            DB      "V"     ;Skip over one character in template
0006            DB      "T"     ;Copy up to specified character
0007            DB      "W"     ;Skip up to specified character
0008            DB      "U"     ;Copy rest of template
0009            DB      "E"     ;Kill line with no change in template (Ctrl-X)
000A            DB      "J"     ;Cancel line and update template
000B            DB      "D"     ;Backspace (same as Ctrl-H)
000C            DB      "P"     ;Enter Insert mode
000D            DB      "Q"     ;Exit Insert mode
000E            DB      "R"     ;Escape sequence to represent escape character
000F            DB      "R"     ;End of table - must be same as a previous byte
```

For example, the character sequence ESC S will copy one character from the template to the new line. The next to last entry in the table is the escape sequence to be used to pass the escape character. In the standard table shown here, this is done by typing ESC R, but it could also be set up for any other escape sequence, including ESC ESC (hitting escape twice).

Customizing the I/O system

In order to provide the user with maximum flexibility, the disk and simple device I/O handlers of MS-DOS are a separate subsystem which may be configured for virtually any real hardware. This I/O system is located starting at absolute address 400 hex, and may be any length. The DOS itself is completely relocatable and normally starts immediately after the I/O system.

Beginning at the very start of the I/O system (absolute address 400 hex) is a series of 3-byte jumps (long intra-segment jumps) to various routines within the I/O system. These jumps and their starting offsets (relative segment 40H) look like this:

```
0000    JMP     INIT    ; System initialization
0003    JMP     STATUS  ; Console status check
0006    JMP     CONIN   ; Console input
0009    JMP     CONOUT  ; Console output
000C    JMP     PRINT   ; Printer output
000F    JMP     AUXIN   ; Auxiliary input
0012    JMP     AUXOUT  ; Auxiliary output
0015    JMP     READ    ; Disk read
0018    JMP     WRITE   ; Disk write
001B    JMP     DSKCHG  ; Return disk change status
001E    JMP     SETDATE ; Set current date
0021    JMP     SETTIME ; Set current time
0024    JMP     GETDATE ; Read time and date
0027    JMP     FLUSH   ; Flush keyboard input buffer
002A    JMP     MAPDEV  ; Device mapping
```

The first jump, to INIT, is the entry point from the system boot. All the rest are entry points for subroutines called by the DOS. Inter-segment calls are used so that the code segment is always 40 hex (corresponding to absolute address 400 hex) with a displacement of 3, 6, 9, etc. Thus each routine must make an inter-segment return when done.

The function of each routine is as follows:

INIT - System initialization

Entry conditions are established by the system bootstrap loader and should be considered unknown. The following jobs must be performed:

A. All devices are initialized as necessary.

B. A local stack is set up, DS:SI are set to point to an initialization table, and DX is set with the number of paragraphs (16-byte units) of total memory. If DX is set 0001, then MS-DOS will perform a memory scan to determine size. Then an inter-segment call is made to the first byte of the DOS, using a displacement of zero. For example:

```
MOV     AX,CS           ; Get current segment
MOV     DS,AX
MOV     SS,AX
MOV     SP,OFFSET STACK
MOV     SI,OFFSET INITTAB
MOV     DX,1            ;Use automatic size determination
CALL    DOSSEG:0
```

The initialization table provides the DOS with information about the disk system. The first entry in the table is one byte with the number of disk I/O drivers, N. This byte is followed by N 3-byte entries, each of which consists of:

1. 1 Byte. The physical drive number this entry refers to.

2. 2 Bytes. The offset of this drive's Drive Parameter Table (DPT) in DS--see below. Similar drives may share a DPT.

Each entry in this table is considered a separate I/O driver, numbered from 0 to N-1. Each physical disk drive may have more than one I/O driver, thus allowing more than one format/density/configuration for each drive. Each drive has only one File Allocation Table in memory, which is equal in size to the largest table needed for any configuration specified for that drive.

For example, if a system has two disk drives, both of which may contain either single or double density diskettes, then the table might look like this:

```
        DB      4               ;4 I/O drivers

        DB      0               ;Drive 0
        DW      SDRIVE          ;Single density DPT
        DB      0               ;Still drive 0
        DW      DDRIVE          ;Double density DPT

        DB      1               ;Repeat it all for drive 1
        DW      SDRIVE
        DB      1
        DW      DDRIVE
```

The Drive Parameter Table, or DPT, has the following entries:

1. SECSIZ. 2 Bytes. The size, in bytes, of the physical disk sector. The minimum value is 32 bytes, the maximum practical value is 16K. This number need not be a power of 2.

2. CLUSSIZ. 1 Byte. The number of sectors in an allocation unit. This number must be a power of 2. This limits it to the values 1, 2, 4, 8, 16, 32, 64, and 128. By making the allocation unit small, less disk space is wasted because the last allocation unit of each file is only half full on the average. By making the allocation unit large, less space is taken up both on the disk and in memory for the File Allocation Table. A good choice is to make the allocation unit approximately equal to the square root of the disk size (to the nearest power of 2). For example, a standard floppy disk with 256K would use an allocation unit of 512 bytes, or 4 physical sectors. A 2D floppy disk with 1K sectors has 1.2 Mbytes, and would use an allocation uni of 1K, or 1 physical sector.

3. RESSEC. 2 Bytes. The number of reserved sectors at the start of the disk. At least one sector is usually reserved for a disk bootstrap loader and more may be reserved to place the I/O system or all of MS-DOS in this reserved area.

4. FATCNT. 1 Byte. The number of File Allocation Tables. This is noramlly two, to provide one backup.

5. MAXENT. 2 Bytes. The number of directory entries. This may be any number less than 4080. For maximum efficiency, however, it should be a multiple of the number of directory entries that can fit in one physical sector, at 32 bytes per directory entry.

6. DSKSIZ. 2 Bytes. The number of physical disk sectors. Being represented with only 16 bits, this number clearly must be less than 64K. If a large disk has more physical sectors than this, the size of the physical sector seen by MS-DOS must be increased by using multiples of the physical secto Every time the I/O system documentation says "physical sector," consider this to mean, for example, two physical sectors. Then the size of this new "physical sector," SECSIZ, is twice as big as before, DSKSIZ is half as big, and the READ and WRITE routines must work in terms of these new sectors.

Below are the Microsoft standard Drive Parameter Tables for the most popular floppy disk formats. The FAT identification byte is placed in the first byte of the FAT when the disk directory is cleared by FORMAT, and may be used by MAPDEV to support mulitple formats. If your format is not listed and you wish to be interchange compatible with other manufacturers, contact Microsoft.

8" IBM 3740 format, singled-sided, single-density, 128 bytes per sector, soft sectored:

```
                DW      128     ;128 bytes/sector
                DB      4       ;4 sectors/allocation unit
                DW      1       ;Reserve one boot sector
                DB      2       ;2 FATs - one for backup
                DW      68      ;17 directory sectors
                DW      77*26   ;Tracks * sectors/track = disk size
```

FAT identification byte is FE hex.

8" Double-sided, double-density, 1024 bytes per sector, soft sectored:

```
                DW      1024
                DB      1
                DW      1
                DB      2
                DW      192
                DW      77*8*2
```

FAT identification byte is FE hex. Multiple sectors are to be transferred by transferring all sectors on side 0 of a track, then all sectors of side 1 on that track, then stepping to the next track. From the beginning of the disk, the order is: Track 0, side 0, sectors 1 - 8; track 0, side 1, sectors 1 - 8; track 1, side 0, sectors 1 - 8; track 1, side 1, sectors 1 - 8; etc.

5" Single-sided, double-density, 512 bytes per sector, soft sectored:

```
                DW      512     ;512 bytes/sector
                DB      1       ;1 sector/allocation unit
                DW      1       ;Reserve one boot sector
                DB      2       ;2 FATs - one for backup
                DW      64      ;4 directory sectors
                DW      40*8    ;Tracks * sectors/track = disk size
```

FAT identification byte is FE hex.

5" Double-sided, double-density, 512 bytes per sector, soft sectored:

```
                DW      512     ;512 bytes/sector
                DB      2       ;2 sectors/allocation unit
                DW      1       ;Reserve one boot sector
                DB      2       ;2 FATs - one for backup
                DW      112     ;7 directory sectors
                DW      40*8*2  ;Tracks * sectors/track * sides = disk size
```

FAT identification byte is FF hex. The sector order is the same as for double-sided, double-density 8" disks above.

5" Double-sided, double-density, double track density, 512 bytes per sector, soft sectored:

```
                DW      512     ;512 bytes/sector
                DB      4       ;2 sectors/allocation unit
                DW      1       ;Reserve one boot sector
                DB      2       ;2 FATs - one for backup
                DW      144     ;9 directory sectors
                DW      80*8*2  ;Tracks * sectors/track * sides = disk size
```

FAT identification byte is FD hex.

C. When the DOS returns to the INIT routine in the I/O system, DS has the segment of the start of free memory, where a program segment has been set up. The remaining task of INIT is to load and execute a program at 100 hex in this segment, normally COMMAND.COM. The steps are:

1. Set the disk transfer address to DS:100H.

2. Open COMMAND.COM. If not on disk, report error.

3. Load COMMAND using the block read function (Function 39). If end-of-file was not reached, or if no records were read, report an error.

4. Set up the standard initial conditions and jump to 100 hex in the new program segment.

```
        MOV     DX,100H
        MOV     AH,26
        INT     21H             ;Set transfer address to DS:100H
        MOV     CX,WORD PTR DS:6  ;Get maximum size of segment
        MOV     BX,DS           ;Save segment for later
; DS must be set to CS so we can point to the FCB
        MOV     AX,CS
        MOV     DS,AX
        MOV     DX,OFFSET FCB   ;File Control Block for COMMAND.COM
        MOV     AH,15
        INT     21H             ;Open COMMAND.COM
        OR      AL,AL
        JNZ     COMERR          ;Error if file not found
        MOV     WORD PTR FCB+14,1  ;Set record length to 1 byte
        MOV     AH,39
        INT     21H             ;Block read
        JCXZ    COMERR          ;Error if no records read
        CMP     AL,1
        JNZ     COMERR          ;Error if not end-of-file
        MOV     DS,BX           ;All segment reg.s must be the same
        MOV     ES,BX
        MOV     SS,BX
        MOV     SP,5CH          ;Stack must be 5C hex
        XOR     AX,AX
        PUSH    AX              ;Put zero of top of stack
        MOV     DX,80H
        MOV     AH,26
        INT     21H             ;Set transfer address to default
        PUSH    BX
        MOV     AX,100H
        PUSH    AX
        RET                     ;FAR return - jump to COMMAND
COMERR:
        MOV     DX,BADCOM
        MOV     AH,9
        INT     21H             ;Print error message
STALL:  JMP     STALL           ;Don't know what to do

BADCOM: DB      13,10,"Bad or missing Command Interpreter",13,10,"$"

FCB:    DB      1,"COMMAND COM"
        DB      25 DUP (0)
```

STATUS - Console input status

If a character is ready at the console, this routine returns with the zero flag cleared and the character in AL, which is still pending. Once a character has been returned with this call, that same character must be returned every time the call is made until a CONIN call is made. In other words, this call leaves the character in the input buffer, and only CONIN can remove it. If no character is ready, the zero flag is set. No registers other than AL may be changed.

CONIN - Console input

Wait for a character from the console, then return with the character in AL. No other registers may be changed.

CONOUT - Console output

Output the character in AL to the console. No registers may be affected.

PRINT - Printer output

Output the character in AL to the printer. No registers may be affected.

AUXIN - Auxiliary input

Wait for a byte from the auxiliary input device, then return with the byte in AL. No other registers may be affected.

AUXOUT - Auxiliary output

Output the byte in AL to the auxiliary output device. No registers may be affected.

READ - Disk read
WRITE - Disk write

On entry,

AL = I/O driver number (starting with zero)
AH = Verify flag (WRITE only) 0=no verify, 1=verify after write
CX = Number of physical sectors to transfer
DX = Logical sector number
DS:BX = Transfer address.

The number of sectors specified are transfered using the given I/O driver at the transfer address. "Logical sector numbers" are obtained by numbering each sector sequentially starting from zero, and continuing across track boundaries. Thus for standard 8" floppy disks, for example, logical sector 0 is track 0 secto 1, and logical sector 53 is track 2 sector 2. This conversion from logical sector number to physical track and sector is done simply by dividing by the numb sectors per track. The quotient is the track number, and the remainder is the sector on that track. (If the first sector on a track is 1 instead of 0, as with standard floppy disks, add one to the remainder.)

"Sector mapping" is not used by this scheme, and is not recommended unless contiguous sectors cannot be read at full speed. If sector mapping is desired, however, it may be done after the logical sector number is broken down into track and sector. The 8086 instruction XLAT is quite useful for this mapping.

All registers except the segment registers may be destroyed by these routines. If the transfer was successfully completed, the routines should return with the carry flag clear. If not, the carry flag should be set, and CX should have the number of sectors remaining to be transfered (including the sector in error). A code for the type of error should be returned in AL, which will be used to print one of the following messages:

AL	Error type
0	Write protect (disk writes only, of course)
2	Not ready
4	Data
6	Seek
8	Sector not found
10	Write fault
12	Disk - This is a catch-all for any other errors

DSKCHG - Disk change test

This routine takes as input a disk drive number in AL and AH is zero. It returns

AH = -1 if disk has been changed.
AH = 0 if it is not known whether the disk has been changed.
AH = 1 if disk could not have been changed.

and AL = I/O driver number to use for this diskette and drive.
Carry flag clear

If this routine requires a disk read and the read is unsuccessful, it should return with carry set and error code in AL (same as READ or WRITE). This will invoke normal hard disk error handling, except the error can not be ignored.

This routine is called whenever a directory search has been made and the disk could legally have been changed. The purpose is to minimize unnecessary re-reading of disk directory information if the disk has not been changed, and to provide configuration information if it has. If, for example, a drive will be required to read both single and double density disks, this routine will make the determination of which format is currently present, and provide the corresponding I/O driver number.

Examining this example more closely, suppose the initialization table appeared as follows:

```
DB      4               ;4 I/O drivers

DB      0               ;Drive 0
DW      SDRIVE          ;Single density DPT
DB      0               ;Still drive 0
DW      DDRIVE          ;Double density DPT

DB      1               ;Repeat it all for drive 1
DW      SDRIVE
DB      1
DW      DDRIVE
```

If a directory search is to be made on drive B, this routine will be called with AH=0, AL=1. If the routine determines that a single-density disk is presently in the drive, it will return with AL=2; if a double density disk, AL=3. If this is a change from the previous density used in this drive, is should also set AH=-1; otherwise, AH=0.

One way to determine density is to simply try to read the disk with the same density as last time; if that doesn't work, switch densities and try again. If neither can be read (after suitable re-tries), the routine should return with the carry flag set and the error code (same as READ or WRITE, above) in AL. Other systems will always have track 0 formatted single density, with a flag indicating what the rest of the disk is formatted like. Again, if a hard disk error occurs attempting to read this information, return the same error indicator as READ or WRITE would.

Eight-inch double sided disks have their index hole punched in a different place from the single-sided disks, and some drives provide a "two-side" status signal to indicate which is being used. This provides an easy way to distinguish format.

If there is a one-to-one mapping between physical disk drives and I/O drivers, then AL may be left unchanged. AH must still return disk change information, if available.

Floppy disk systems with no way to know if the disk has been changed will simply return AH = 0 whenever this routine is called. Some floppy disk drives provide a disk change signal, which simply latches the fact that the drive door has been opened since the last disk access. Another way to tell is if the head of the drive is still loaded from the last disk command, then one may assume the disk has not been changed. (In this case, the head not loaded does not mean the disk has been changed, it means unknown.) A non-removable hard disk should always return that disk is not changed.

SETDATE - Set date

On entry, AX has the count of days since January 1, 1980. If the system has time-keeping hardware, the date should roll over at midnight. Otherwise, it should simply be stored for return by GETDATE.

SETTIME - Set time

On entry, CX and DX have the current time:

CH = hours (0-23)
CL = minutes (0-59)
DH = seconds (0-59)
DL = hundredths of seconds (0-99)

Each of these is a binary number that has been checked for proper range. If time-keeping hardware is not used, the time should simply be stored for return by GETDATE.

GETDATE - Read date and time

Returns the following information:

AX = count of days since 1-1-80.
CH = hours
CL = minutes
DH = seconds
DL = hundredths of seconds

No other registers may be affected.

FLUSH - Flush keyboard buffer

If the console input keyboard has a hardware or software type-ahead buffer, the buffer should be cleared with this call. If there is no buffer, this routine should simply return.

MAPDEV - Map disk I/O drivers

This routine can be used to map physical disk drives with their I/O drivers. It is called AFTER the File Allocation Table is read (which is after the DSKCHG call), which means that DSKCHG must have returned an I/O driver which could properly read the disk, and for which the File Allocation Tables are the same number of sectors and in the same place on the disk. Then, the first byte of the FAT is used to determine the rest of the disk format. This byte may legally be in the range 0F8 hex to 0FF hex, and is normally set at format time.

On entry,

AL = I/O driver used to read the FAT
AH = First byte of FAT (range F8 to FF)

on exit,

AL = I/O driver for this diskette and drive.

This routine is particularly suited for distinguishing between double-sided and single-sided disks. For example, the double-sided drive might use an allocation unit twice as large as the single-sided, so the allocation table will be the same size. The first byte of the FAT could be FF for single-sided, FE for double sided. The I/O driver for double sided would use an initialization table with more directory entries and more sectors; the driver itself could interleave sides of the disk between stepping the head, provided all of the FATs fit in one track. DSKCHG could return the I/O driver for the single-sided disks, which would be adequate for reading the FAT from double-sided disks. Then MAPDEV could use the least significant bit of the first byte of the FAT to return the correct I/O driver.

The advantage of using MAPDEV over returning the completely correct I/O driver in DSKCHG is that their are no extra disk accesses, since the FAT will be read anyway.

In most systems, the entire input range F8 to FF will not be meaningful. This routine, however, should always return a valid I/O driver number of the drive.

MS-DOS DISK CONFIGURATION AND BOOTSTRAP LOADING

MS-DOS disks are divided into four areas:

1. Reserved
2. File allocation tables
3. Directory
4. File data

The size of the reserved area is specified by the OEM and should be as small as possible. Normally, only one sector is needed for a bootstrap loader. In systems where the first track is formatted single density while the rest of the disk is double density, it may be simplest to include the entire first track in the reserved area. Sectors in the reserved area need not be the same size as the sectors on the rest of the disk since they are never accessed by the file system.

The size of the File Allocation Tables and the directory are computed during initialization from the OEM's Drive Parameter Table. The size of the data area is simply everything that's left, truncated to whole Allocation Units. (Any sectors so truncated are never used.)

MS-DOS and the OEM's I/O system reside in the data area of the disk. They are each in their own file, properly recorded in both the directory and the File Allocation Table. However, in order to simplify bootstrap loading of these files, they can be guaranteed to be in fixed locations on the disk, on consecutive sectors. Specifically, the file IO.SYS always starts on the first sector of the data area. The file MSDOS.SYS always starts on the first allocation unit immediately after IO.SYS. Thus the bootstrap loader need only deal with loading consecutive sectors beginning at a fixed location on the disk.

In order to ensure these .SYS files are in their proper locations, the files are hidden from all ordinary directory operations by an attribute bit in the directory. This means the files cannot be seen with the DIR command nor copied with the COPY command. Instead, the program SYS.COM is provided to allow copying these files from disk to disk. SYS will only perform the copy if either:

1) The destination disk has no files on it (this is the basic requirement for locating the .SYS files in the right place).

2. The destination already has both .SYS files (which are assumed to be in the right place, so the copy operation will just overwrite them).

The primary purpose of for putting MS-DOS and the I/O system in the data area is to allow "system disks", from which MS-DOS can be loaded, and "data disks", which have more data space. It also allows the size of MS-DOS or the I/O system to change, instead of locking them into a fixed size reserved area. (NOTE: If either MS-DOS or the I/O system grow to exceed the number of allocation units they have been assigned on the disk, then previous system disks can NOT be updated with the larger version. The solution for the user is to create a new system disk, and copy files to it. The old system disk may then be used to load an old system, or it may be used as a data disk. This is the price paid to have a simple bootstrap loader for consecutive sectors.)

Writing the bootstrap loader requires knowing where the data area starts, since IO.SYS is the first thing in the data area. Here are the starting locations for Microsoft standard formats:

FORMAT	Sector number dec	hex	track,side,sector
8" single side, single density	30	1E	1,0,5
8" double side, double density	11	0B	0,1,4
5" single side, double density	7	7	0,0,8
5" double side, double density	10	0A	0,1,3

If you are not using one of Microsoft's standard formats, you can figure out the start of the data area using the drive initialization table. The approach is simply to determine the size of each component preceding the data area, and add it up.

First, the size of the reserved area. This appears directly in the initialization table.

Next the size of the directory. Divide the sector size by 32 to find the number of entries per sector. Divide this result into the number of directory entries, rounding up if there is any remainder. This is the number of directory sectors.

The number of sectors in the File Allocation Tables depends on the size of the data area, which in turn depends on the size of the FAT. Start by assuming a FAT size of one sector. Compute the start of the data area with

```
[ (FAT size) * (number of FATs) ] + (number of directory sectors) +
        (number of reserved sectors) = start of data area
```

then figure the size of the data area with

(size of disk) - (start of data area) = size of data area.

The number of allocation units on the disk is what actually determines the size of the FAT. This is simply

(size of data area) / (sectors per allocation unit) = number of allocation units

Each allocation unit requires 1.5 bytes in the FAT, plus three extra bytes are needed because allocation units 0 and 1 are reserved.

[(number of allocation units) * 1.5] + 3 = FAT size (round up if not integer)

This is a good estimate of the FAT size, but it is still only an estimate. Now go back and do it all over again, except this time when computing the size of the data area, use this estimate of FAT size instead of 1. This re-computation should be repeated until the estimate of FAT size is the same twice in a row.

If the final calculation of the number of allocation units (a division) results in a remainder, these are sectors that will go completely unused, since there are not enough to make a whole allocation unit. To prevent this from being a total waste, the number of sectors in the directory can be adjusted so there are just enough sectors left to fill out the last allocation unit. For example, the initial selection for single density 8" disks was 64 directory entries. This, however, leaves one sector unused; so instead, one sector was added to the directory, and the allocation units come out with an exact number of sectors. This added sector in the directory is still not used very often, but it is available if needed.

```
        TITLE   IOSYS  - - Skeleton IO.SYS for MSDOS
;************************************************************************
;
;
; I/O System for MSDOS version 1.20 and later.
;
;
;************************************************************************

BIOSSEG SEGMENT AT 40H

DOSSTART EQU    0800H           ;Position of DOS after BIOS on disk
DOSSIZE EQU     2000H           ;Max size of DOS

        ASSUME  CS:BIOSSEG

        JMP     NEAR PTR INIT
        JMP     NEAR PTR CONSTAT
        JMP     NEAR PTR CONIN
        JMP     NEAR PTR CONOUT
        JMP     NEAR PTR PRINT
        JMP     NEAR PTR AUXIN
        JMP     NEAR PTR AUXOUT
        JMP     NEAR PTR READ
        JMP     NEAR PTR WRITE
        JMP     NEAR PTR DSKCHG
        JMP     NEAR PTR SETDATE
        JMP     NEAR PTR SETTIME
        JMP     NEAR PTR GDATTIM
        JMP     NEAR PTR FLUSH
        JMP     NEAR PTR MAPDEV

BADCOM: DB      13,10,"Error in loading Command Interpreter",13,10,"$"
FCB:    DB      1,"COMMAND COM"
        DB      25 DUP(0)

INIT    PROC    FAR
        XOR     BP,BP           ; Set up stack just below I/O system.
        MOV     SS,BP
        MOV     SP,800H         ;BIOSEG offsetted from standard.
        PUSH    CS
        POP     DS
        PUSH    CS
        POP     ES

        ASSUME  DS:BIOSSEG, ES:BIOSSEG

;Perform initialization of all hardware here

        MOV     SI,DOSSTART
        MOV     DI,OFFSET ENDBIOS
        MOV     CX,DOSSIZE/2
        REP     MOVSW
        MOV     SI,OFFSET INITTAB
        MOV     DX,1            ;Cause auto top of memory scan.
        CALL    FAR PTR MSDOS
        MOV     DX,100H
```

```
        MOV     AH,26           ;Set DMA address
        INT     33
        MOV     CX,DS:6         ;Get size of segment
        MOV     BX,DS           ;Save segment for later

; DS must be set to CS so we can point to the FCB.

        PUSH    CS
        POP     DS
        MOV     DX,OFFSET FCB   ;File Control Block for COMMAND.COM
        MOV     AH,15
        INT     33              ;Open COMMAND.COM
        OR      AL,AL
        JNZ     COMERR          ;Error if file not found
        MOV     WORD PTR FCB+14,1 ;Set record length field
        MOV     AH,39           ;Block read (CX already set)
        INT     33
        JCXZ    COMERR          ;Error if no records read
        TEST    AL,1
        JZ      COMERR          ;Error if not end-of-file

; Make all segment registers the same.

        MOV     DS,BX
        MOV     ES,BX
        MOV     SS,BX
        MOV     SP,5CH          ;Set stack to standard value
        XOR     AX,AX
        PUSH    AX              ;Put zero on top of stack for return
        MOV     DX,80H
        MOV     AH,26
        INT     33              ;Set default transfer address (DS:0080)
        PUSH    BX              ;Put segment on stack
        MOV     AX,100H
        PUSH    AX              ;Put address to execute within segment on stack
        RET                     ;Jump to COMMAND
INIT    ENDP

COMERR:
        MOV     DX,OFFSET BADCOM
        MOV     AH,9            ;Print string
        INT     33
        STI
STALL:  JMP     STALL

        PAGE
```

```
        ASSUME  DS:NOTHING, ES:NOTHING

;
; Temporary storage for Date and Time Variables.
;

DATE    DW      0000
TIMECX  DW      0000
TIMEDX  DW      0000

;
; Routine to retrieve Date and Time from storage.
;
; On exit:
;       AX = Count of days since January 1, 1980
;       CH = Hours
;       CL = Minutes
;       DH = Seconds
;       DL = Hundreths of seconds
;

GDATTIM PROC    FAR
        MOV     AX,DATE
        MOV     CX,TIMECX
        MOV     DX,TIMEDX
        RET
GDATTIM ENDP

;
; Routine to set the time.
;
;  On entry:
;
;       CH = Hours (0 => 23)
;       CL = Minutes (0 => 59)
;       DH = Seconds (0 => 59)
;       DL = Hundredths of seconds (0 => 99)
;

SETTIME PROC    FAR
        MOV     TIMECX,CX
        MOV     TIMEDX,DX
        RET
SETTIME ENDP

;
; Routine to set the date.
;
;  On entry:
;
;       AX = Count of days since January 1, 1980
;

SETDATE PROC    FAR
        MOV     DATE,AX
        RET
SETDATE ENDP

        PAGE
```

```
;
; Routine to retrieve console status and snap of character.
;
;  On exit:
;       AL = Copy of character waiting in buffer.
;       Z  = Non-Zero if character is waiting.
;
;       Z  = Zero if no character is waiting.
;
;            No registers beside AL may be used.  The routine
;            must be able to return a copy of the same character
;            waiting in the buffer, until the character is actually
;            read with CONIN.
;
CONSTAT PROC    FAR

        RET
CONSTAT ENDP

;
; Routine to retrieve the character from the console buffer.
;
;  On exit:
;       AL = The character waiting in the buffer.
;
;            Routine does not return until a valid character
;            is available.  No other registers can be used
;            except the AL.
;
CONIN   PROC    FAR

        RET
CONIN   ENDP

;
; Routine to flush any type a head characters from Console input
;   buffer.
;
;  On exit:
;       No registers may be changed.
;
FLUSH   PROC    FAR
        RET
FLUSH   ENDP
```

```
;
; Routine to output a character to the console.
;
;  On entry:
;       AL = Character to be output
;
;            No registers may be changed.
;
CONOUT  PROC    FAR

        RET
CONOUT  ENDP

        PAGE

;
; Routine to output a character to the printer.
;
;  On entry:
;       AL = Character to be output.
;
;            No registers may be changed.
;
PRINT   PROC    FAR
        RET
PRINT   ENDP

;
; Routine to read a character from the auxilary port.
;
;  On exit:
;       AL = Character.
;
;            No other registers may be changed.
;
AUXIN   PROC    FAR
        RET
AUXIN   ENDP

;
; Routine to send a character to the auxilary port.
;
;  On entry:
;       AL = Character to send.
;
;            No registers may be changed.
;
AUXOUT  PROC    FAR
        RET
AUXOUT  ENDP
```

```
; Disk change function.
; On entry:
;       AL = disk drive number.
; On exit:
;       AH = -1 (FF hex) if disk is changed.
;       AH = 0 if don't know.
;       AH = 1 if not changed.
;
;       CF clear if no disk error.
;       AL = disk I/O driver number.
;
;       CF set if disk error.
;       AL = disk error code (see disk read below).

DSKCHG  PROC    FAR

        RET

DSKCHG  ENDP

        PAGE
```

```
;Map Disk I/O drivers
;
;  On entry:
;       AL = I/O driver used to read the FAT
;       AH = First byte of FAT (range F8 to FF)
;
;  On exit:
;       AL = I/O driver for this drive.
;

MAPDEV  PROC    FAR

        RET

MAPDEV  ENDP

        PAGE
```

```
; Disk read function.
;
; On entry:
;       AL = Disk I/O driver number
;       BX = Disk transfer address in DS
;       CX = Number of sectors to transfer
;       DX = Logical record number of transfer
; On exit:
;       CF clear if transfer complete
;
;       CF set if hard disk error.
;       CX = number of sectors left to transfer.
;       AL = disk error code
;               0 = write protect error
;               2 = not ready error
;               4 = CRC error
;               6 = seek error
;               8 = sector not found
;              10 = write fault
;              12 = "data error" (any other error)

READ    PROC    FAR

        RET
READ    ENDP

        PAGE
```

```
; Disk write function.
;
; On entry:
;       AL = Disk I/O driver number
;       BX = Disk transfer address in DS:
;       CX = Number of sectors to transfer.
;       DX = Logical Record number of transfer.
;
; On exit:
;       CF = Clear if transfer completed.
;       CF = Set if Hard disk error.
;       CX = number of sectors left to transfer.
;       AL = Disk error code:
;               0 = Write protect error.
;               2 = Not ready error.
;               4 = CRC error.
;               6 = Seek error.
;               8 = Sector not found.
;               10= Write fault.
;               12= Data error. (catch all)

WRITE   PROC    FAR

        RET
WRITE   ENDP

        PAGE
```

```
;
; Generalized error handler for a paticular status word that
;   may be returned by a disk controller chip.
;
ERROR   PROC    FAR
        MOV     BL,-1
        MOV     CS:[DI],BL      ; Indicate we don't know where head is.
        MOV     SI,OFFSET ERRTAB
GETCOD:
        INC     BL              ; Increment to next error code.
        LODS    CS:BYTE PTR [SI]
        TEST    AH,AL           ; See if error code matches disk status.
        JZ      GETCOD          ; Try another if not.
        MOV     AL,BL           ; Now we've got the code.
        SHL     AL,1            ; Multiply by two.
        STC
        RET
ERROR   ENDP

ERRTAB:                         ;  Some sample status bits
        DB      40H             ;Write protect error
        DB      80H             ;Not ready error
        DB      8               ;CRC error
        DB      2               ;Seek error
        DB      10H             ;Sector not found
        DB      20H             ;Write fault
        DB      7               ;Data error

        PAGE
```

```
; Function:
;       Seeks to proper track.
; On entry:
;       Same as for disk read or write above.
; On exit:
;       AH = Drive select byte
;       DL = Track number
;       DH = Sector number
;       SI = Disk transfer address in DS
;       DI = pointer to drive's track counter in CS
;       CX unchanged (number of sectors)

SEEK:

        RET

        PAGE
```

```
;
; MSDOS drive initialization tables and other what not.
;
; Some example drives are shown.
;
;  Drives 0 and 1 are:
;               Single Density 8-inch 26 sector drives. (256,256 bytes)
;               or
;               Double Density / Double Sided 8-inch 8 sector drives.
;               (1,261,568 bytes)
;
;  Drive 2 is a:
;               8-inch Winchester with 256 cylinders, 4 heads and formatted
;               for 17 sectors of 512 bytes per track. (8,912,896 bytes)
;
;  Drive 3 is a:
;               Section of main memory to use as a high speed RAM drive.
;               40000 hex bytes (262,144 bytes) broken up into 512 byte
;               sectors, with 8 sectors per track and 64 tracks per RAM
;               drive.
;

INITTAB DB      6
        DB      0
        DW      LSDRIVE
        DB      0
        DW      LDDRIVE
        DB      1
        DW      LSDRIVE
        DB      1
        DW      LDDRIVE
        DB      2
        DW      HDDRIVE         ;Hard disk.
        DB      3
        DW      MDDRIVE         ;Memory drive.


DDRIVE  STRUC
BYTSEC  DW      0000            ;Bytes / sector.
SECALLC DB      00              ;Sectors / Allocation Unit.
RESSEC  DW      0000            ;Reserved sectors.
NUMFAT  DB      00              ;Number of FAT's
NUMDIR  DW      0000            ;Number of directory entries.
TOTSEC  DW      0000            ;Total Number of sectors.
DDRIVE  ENDS

LSDRIVE DDRIVE  <128,4,2,2,68,2002>

LDDRIVE DDRIVE  <1024,1,0,2,192,1232>

HDDRIVE DDRIVE  <512,2,0,2,1024,17408>

MDDRIVE DDRIVE  <512,1,0,2,68,512>

        PAGE
```

```
ENDBIOS LABEL   BYTE
BIOSSEG ENDS

DOSSEG  SEGMENT
MSDOS   LABEL   FAR
DOSSEG  ENDS
        END
```

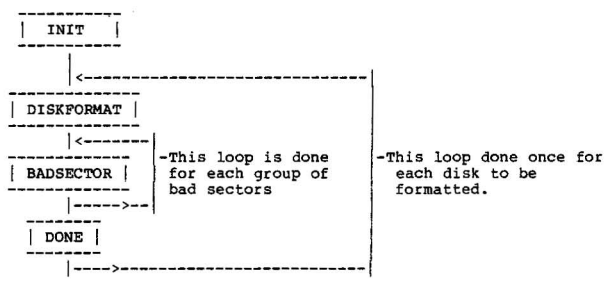
FORMAT - formats a new disk, clears the FAT and DIRECTORY and optionally copies the SYSTEM and COMMAND.COM to this new disk.

Command syntax:

```
FORMAT  [drive:][/switch1][/switch2]...[/switch16]
```

Where "drive:" is a legal drive specification and if omitted indicates that the default drive will be used. There may be up to 16 legal switches included in the command line.

The OEM must supply four (NEAR) routines to the program along with 4 data items. The names of the routines are INIT, DISKFORMAT, BADSECTOR, and DONE, and their flow of control (by the Microsoft module) is like this:



The INIT, DISKFORMAT, and BADSECTOR routines are free to use any MS-DOS system calls, except for file I/O and FAT pointer calls on the disk being formatted. DONE may use ANY calls, since by the time it is called the new disk has been formatted.

The following data must be declared PUBLIC in a module provided by the OEM:

SWITCHLIST - A string of bytes. The first byte is count N, followed by N characters which are the switches to be accepted by the command line scanner. Alphabetic characters must be in upper case. The switch to indicate that you want a system transferred, normally "S", must be the last switch in the list. Up to 16 switches are permitted. Normally a "C" switch is specified for "Clear". This switch should cause the formatting operation to be bypassed (within DISKFORMAT or BADSECTOR). This is provided as a time-saving convenience to the user, who may wish to "start fresh" on a previosly formatted and used disk.

FATID - BYTE location containing the value to be used in the first byte of the FAT. Must be in the range F8 hex to FF hex. This byte may be used to differentiate between various formats for the same physical drive (like single or double sided).

STARTSECTOR - WORD location containing the sector number of the first sector of the data area.

FREESPACE - WORD location which contains the address of the start of free memory space. This is where the system will be loaded, by the Microsoft module, for transferring to the newly formatted disk. Memory should be available from this address to the end of memory, so it is typically the address of the end of the OEM module.

The following routines must be declared PUBLIC in the OEM-supplied module:

INIT - An initialization routine. This routine is called once at the start of the FORMAT run after the switches have been processed. This routine should perform any functions that only need to be done once per FORMAT run. An example of what this routine might do is read the boot sector into a buffer so that it can be transferred to the new disks by DISKFORMAT. If this routine returns with the CARRY flag set it indicates an error, and FORMAT will print "Fatal format error" and quit. This feature can be used to detect conflicting switches (like specifying both single and double density) and cause FORMAT to quit without doing anything.

DISKFORMAT - Formats the disk according to the options indicated by the switches and the value of FATID must be defined when it returns (although INIT may have already done it). This routine is called once for EACH disk to be formatted. If neccessary it must transfer the Bootstrap loader. If any error conditions are detected, set the CARRY flag and return to FORMAT. FORMAT will report a 'Format failure' and prompt for another disk. (If you only require a clear directory and FAT then simply setting the appropriate FATID, if not done by INIT, will be all that DISKFORMAT must do.)

BADSECTOR - Reports the sector number of any bad sectors that may have been found during the formatting of the disk. This routine is called at least once for EACH disk to be formatted, and is called repeatedly until AX is zero or the carry flag is set. The carry flag is used just as in DISKFORMAT to indicate an error, and FORMAT handles it in the same way. The first sector in the data area must be in STARTSECTOR for the returns from this routine to be interpreted correctly. If there are bad sectors, BADSECTOR must return a sector number in in register BX, the number of consecutive bad sectors in register AX, and carry clear. FORMAT will then process the bad sectors and call BADSECTOR again. When BADSECTOR returns with AX = 0 this means there are no more bad sectors; FORMAT clears the directory and goes on to DONE, so for this last return BX need not contain anything meaningful.

FORMAT processes bad sectors by determining their corresponding allocation unit and marking that unit with an FF7 hex in the File Allocation Table. CHKDSK understands the FF7 mark as a flag for bad sectors and accordingly reports the number of bytes marked in this way.

NOTE: Actual formatting of the disk can be done in BADSECTOR instead of DISKFORMAT on a "report as you go" basis. Formatting goes until a group of bad sectors is encountered, BADSECTOR then reports them by returning with AX and BX set. FORMAT will then call BADSECTOR again and formatting can continue.

DONE - This routine is called after the formatting is complete, the disk directory has been initialized, and the system has been transferred. It is called once for EACH disk to be formatted. This gives the chance for any finishing-up operations, if needed. If the OEM desires certain extra files to be put on the diskette by default, or according to a switch, this could be done in DONE. Again, as in BADSECTOR and DISKFORMAT, carry flag set on return means an error has occurred: 'Format failure' will be printed and FORMAT will prompt for another disk.

The following data is declared PUBLIC in Microsoft's FORMAT module:

SWITCHMAP - A word with a bit vector indicating what switches have been included in the command line. The correspondence of the bits to the switches is determined by SWITCHLIST. The right-most (highest-addressed) switch in SWITCHLIST (which must be the system transfer switch, normally "S") corresponds to bit 0, the second from the right to bit 1, etc. For example, if SWITCHLIST is the string "5,'AGI2S'", and the user specifies "/G/S" on the command line, then bit 4 will be 0 (A not specified), bit 3 will be 1 (G specified), bits 2 and 1 will be 0 (neither I nor 2 specified), and bit 0 will be 1 (S specified).

Bit 0, the system transfer bit, is the only switch used in Microsoft's FORMAT module. This switch is used 1) after INIT has been called, to determine if it is necessary to load the system; 2) after the last BADSECTOR call, to determine if the system is to be written. INIT may force this bit set or reset if desired (for example, some drives may never be used as system disk, such as hard disks). After INIT, the bit may be turned off (but not on, since the system was never read) if something happens that means the system should not be transferred.

After INIT, a second copy of SWITCHMAP is made internally which is used to restore SWITCHMAP for each disk to be formatted. FORMAT itself will turn off the system bit if bad sectors are reported in the system area; DISKFORMAT and BADSECTOR are also allowed to change the map. However, these changes affect only the current disk being formatted, since SWITCHMAP is restored after each disk. (Changes made to SWITCHMAP by INIT do affect ALL disks.)

DRIVE - A byte containing the drive specified in the command line. 0=A, 1=B, etc.

Once the OEM-supplied module has been prepared, it must linked with Microsoft's FORMAT.OBJ module. If the OEM-supplied module is called OEMFOR.OBJ, then the following linker command will do:

```
LINK FORMAT+OEMFOR;
```

This command will produce a file called FORMAT.EXE. FORMAT has been designed to run under MS-DOS as a simple binary .COM file. This conversion is performed by EXE2BIN with the command

```
EXE2BIN FORMAT .COM      [Note the space between "FORMAT" and ".COM"]
```

which will produce the file FORMAT.COM. (If the ".COM" had been omitted, the result would have been named FORMAT.BIN.) FORMAT.COM should be ready to run.

```
;****************************************
;
;         A Sample OEM module
;
;****************************************

CODE    SEGMENT BYTE PUBLIC 'CODE'      ;This segment must be named CODE
                                        ;And it must be PUBLIC
                                        ;And it's classname must be 'CODE'

        ASSUME  CS:CODE,DS:CODE,ES:CODE

;Must declare data and routines PUBLIC

        PUBLIC  FATID,STARTSECTOR,SWITCHLIST,FREESPACE
        PUBLIC  INIT,DISKFORMAT,BADSECTOR,DONE

;This data defined in Microsoft-supplied module

        EXTRN   SWITCHMAP:WORD,DRIVE:BYTE

INIT:

;Read the boot sector into memory
        CALL    READBOOT
        .
        .
;Set FATID to double sided if "D" switch specified
        TEST    SWITCHMAP,4
        JNZ     SETDBLSIDE
        .
        .
        RET

DISKFORMAT:
        .
        .

;Use the bit map in SWITCHMAP to determine what switches are set
        TEST    SWITCHMAP,2             ;Is there a "/C"?
        JNZ     CLEAR                   ;Yes -- clear operation requested
                                        ;  jump around the format code
        .
        < format the disk >
        .
CLEAR:
        .
        .
;Transfer the boot from memory to the new disk
        CALL    TRANSBOOT
        .
        .
        RET
```

```
;Error return - set carry

ERRET:
        STC
        RET

BADSECTOR:
        .
        .
        .
        RET

DONE:
        .
        .
        .
        RET


FATID           DB      0FEH            ;Default Single sided

STARTSECTOR     DW      9

SWITCHLIST      DB      3,"DCS"         ;"S" must be the last switch in the list

FREESPACE       DW      ENDBOOT

BOOT            DB      BOOTSIZE DUP(?) ;Buffer for the boot sector
ENDBOOT         LABEL   BYTE

CODE    ENDS
        END
```

RDCPM:

Reads a file from a disk that has been formatted with a CP/M compatible system and copies that file to a different drive assumed to be formatted with MS-DOS.

COMMAND SYNTAX:

The RDCPM command is similar to the COPY command except that the source file is assumed to be on a CP/M disk:

```
RDCPM filel
RDCPM filel drive
RDCPM filel file2
```

A special form of RDCPM parameters allows the directory of the CP/M disk to be displayed:

```
RDCPM DIR filel
RDCPM DIR drive
```

Wildcard characters ("*" and "?") are acceptable in file names.

HOW RDCPM WORKS:

RDCPM is designed to combine with the CP/M Bios and essentially sits in the space usually used by the CP/M operating system itself.
RDCPM can be be build using an existing CP/M Bios with a minor modification or with a new CP/M Bios that contains a subset of the calls required by the CP/M operating system.

WITH AN EXISTING BIOS:

If you have a CP/M Bios for CP/M-86 then only a slight change to your existing Bios is needed. The GETSEGB call which returns with the address of the Memory Region Table in BX must be modified. The first 16-bit entry in that table must contain the size of your Bios in paragraphs. The calculation is simple:

```
(Size of the Bios in bytes +15)/16
```

This information can be patched into your existing Bios by finding the location of the Memory Region Table and putting this value in the first entry.

RDCPM expects this information in the memory table rather than the usual information required by CP/M.

You must then append this modified Bios to the RDCPM program at location 2500 hex. In order to do this you would load your Bios with the debugger and be sure to remember the size of that program in bytes (register CX contains the size of the program). Then name RDCPM.COM and load it into memory. Change register CX to reflect the size of the Bios and then write RDCPM.COM back out to the disk. You should now have a RDCPM.COM that is executable. See the sequence of debugger commands under "BUILDING RDCPM".

WITH A NEW BIOS:

If you do not have a CP/M Bios written for the 8086 then you must write a Bios to use with RDCPM. You only need to include the following routines:

SELDSK	-	Select a disk drive
SETTRK	-	Set track number
SETSEC	-	Set sector number
SETDMA	-	Set DMA offset address
READ	-	Read selected sector
SECTRAN	-	Sector translate
SETDMAB	-	Set DMA segment address
GETSEGB	-	Get memory table offset

However you must set up the Bios jump vector as if you were writing a full-blown CP/M Bios so that RDCPM will be calling the correct routines for both an existing Bios and a new Bios.

Your CP/M Bios should begin at location 2500H.
A typical CP/M Jump Vector looks like this:

```
ORG     2500H

JMP     INIT
JMP     WBOOT
JMP     CONIN
JMP     CONOUT
JMP     LIST
JMP     PUNCH
JMP     READER
JMP     HOME
JMP     SELDSK          ;********** Need to include this routine
JMP     SETTRK          ;********** Need to include this routine
JMP     SETSEC          ;********** Need to include this routine
JMP     SETDMA          ;********** Need to include this routine
JMP     READ            ;********** Need to include this routine
JMP     WRITE
JMP     LISTST
JMP     SECTRAN         ;********** Need to include this routine
JMP     SETDMAB         ;********** Need to include this routine
JMP     GETSEGB         ;********** Need to include this routine
JMP     GETIOB
JMP     SETIOB
```

BRIEF DESCRIPTION of CP/M BIOS ROUTINES for USE with RDCPM:

SELDSK

The disk specified in CL is selected for reading or writing. (0=A,1=B,2=C,etc.) The bios saves this information for the next disk I/O operation. On return BX contains the base address of the Disk Parameter Header for the selected drive. For an example of the disk parameter information used by CP/M see "DISK DEFINITION TABLES". If an attempt is made to reference a non-existent drive then BX=0.

SETTRK

Register CX contains the track number for the next disk access. The bios saves this information for the next disk I/O operation.

SETSEC

Register CX contains the translated sector number for the next disk access. The bios saves this information for the next disk I/O operation.

SETDMA

Register CX contains the DMA address for the next I/O operation. This address is the offset from the segment address specified in the SETDMAB bios call. The bios saves this information for the next disk I/O operation.

READ

Reads one sector based on the information saved by the previous calls to SELDSK, SETTRK, SETSEC, SETDMA, and SETDMAB.
On return:

AL=0 if no errors occurred.
AL=1 if a non-recoverable error occurred.

SECTRAN

Translates a logical sector number to the appropriate physical sector based on the skew factor of the CP/M system. Register CX contains a logical sector number. Register DX contains the address of the translate table (as returned by the previous call to SELDSK). If DX=0 then no translation takes place otherwise on return BX contains the translated sector number.

SETDMAB

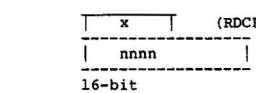
Register CX contains the segment base of the DMA address. The bios saves this information for the next disk I/O operation.

GETSEGB

Returns with the address of the Memory Region table in BX.

The table should contains the following information for RDCPM:

8-bit



16-bit

DISK DEFINITION TABLES:

For a standard 8" single density diskette, the disk definition tables would contains the following information:

Disk Parameter Header

XLT	x	x	x	x	DPB	x	x
16b	16b	16b	16b	16b	16b	16b	16b

Where:

XLT is the offset of the logical to physical TRANSLATION TABLE.

DPB is the offset of the DISK PARAMETER BLOCK for the selected drive.

None of the other values in the table are significant as far as RDCPM is concerned.

The DISK PARAMTER BLOCK is defined as follows for a standard 8" single density diskette:

```
SPT     DW      26              ;Sectors per track
BSH     DB      3               ;Block shift [log2(records/block)]
BSM     DB      7               ;Block mask (record/block-1)
EXM     DB      0               ;Extent size/16k-1
DSM     DW      242             ;Number of blocks on drive-1
DRM     DW      63              ;Number of directory entries-1
        DB      ?               ;Used by CP/M operating system
        DB      ?
        DW      ?
OFF     DW      2               ;No. of reserved tracks at start of disk
```

The logical to physical TRANSLATION TABLE is:

```
XLT     DB      1,7,13,19       ;"Skew factor" of 6
        DB      25,5,11,17
        DB      23,3,9,15
        DB      21,2,8,14
        DB      20,26,6,12
        DB      18,24,4,10
        DB      16,22
```

Once you have written your mini-Bios you must append it to RDCPM.COM in order to produce a working version of RDCPM.

BUILDING RDCPM:

Using the MS-DOS debugger you will append the CP/M Bios to the main RDCPM module at location 2500H.

```
A:DEBUG <cpmbios>

Debug Version n.nn

-R
<register display -- Register CX contains the
size of your Bios file -- remember this number>
-N RDCPM.COM
-L
-R CX
<register display>
<enter remembered value of CX>
-W
-Q

A:
```

A SAMPLE BIOS for USE with RDCPM:

```
CODE    SEGMENT

ASSUME  CS:CODE,DS:CODE,ES:CODE,SS:CODE

ZERO    EQU     $
        ORG     2500H

        JMP     RETURN          ;These "JMP RETURN" are just place holders
        JMP     RETURN
        JMP     RETURN
        JMP     RETURN
        JMP     RETURN
        JMP     RETURN
        JMP     RETURN
        JMP     RETURN
        JMP     RETURN
        JMP     SELDSK
        JMP     SETTRK
        JMP     SETSEC
        JMP     SETDMA
        JMP     READ
        JMP     RETURN
        JMP     RETURN
        JMP     SECTRAN
        JMP     SETDMAB
        JMP     GETSEGT

SELDSK:
        .
        .
        .
        RET

SETTRK:
        .
        .
        .
        RET

SETSEC:
        .
        .
        .
        RET
```

```
SECTRAN:
        .
        .
        .
        RET

SETDMA:
        .
        .
        .
        RET

SETDMAB:
        .
        .
        .
        RET

GETSEGT:
        .
        .
        .
        RET

READ:
        .
        .
        .

RETURN:
        RET
```

```
;CP/M Tables for standard 8" diskettes

SEGTAB    DB        1
MEMCOUNT  DW        ?

DPBASE    LABEL     WORD

DPE0      DW        XLT,0                    ;Drive "A"
          DW        0,0
          DW        0,DPB
          DW        0,0
DPE1      DW        XLT,0                    ;Drive "B"
          DW        0,0
          DW        0,DPB
          DW        0,0

DPB       DW        26
          DB        3
          DB        7
          DB        0
          DW        242
          DW        63
          DB        192
          DB        0
          DW        16
          DW        2

XLT       DB        1,7,13,19
          DB        25,5,11,17
          DB        23,3,9,15
          DB        21,2,8,14
          DB        20,26,6,12
          DB        18,24,4,10
          DB        16,22


CODESIZ   EQU       $-ZERO-2500H
CPMBIOSIZ EQU       (CODESIZ+15)/16

CODE      ENDS
END
```

EXE2BIN - Convert files from EXE format to binary

Command syntax:

```
EXE2BIN  filespec  [d:][filename][.ext]
```

The first parameter is the input file; if no extension is given, it will default to .EXE. The second parameter is the output file. If no drive is given, the drive of the input file is used; if no filename is given, the filename of the input file is used; if no extension is given, .BIN is used.

The input must be in valid EXE format produced by the linker. The "resident", or actual code and data part of the file, must be less than 64K. There must be no STACK segment. Two kinds of conversion are possible depending on the specified initial CS:IP. 1) If CS:IP is not specified, a pure binary conversion is assumed. If segment fix-ups are necessary, the user will be prompted for the fix-up value. 2) If CS:IP is specified as 100H, then it is assumed the file is to be run as a COM file ORGed at 100H, and the first 100H of the file will be deleted. No segment fix-ups are allowed, as COM files must be segment relocatable.

If CS:IP does not meet one of these criteria, or meets the COM file criterium but has segment fix-ups, the error message "File cannot be converted" will be displayed.

Note that to produce standard COM files with the assembler, one must both ORG the file at 100H and specify the first location as the start address (this is done in the END statement). For example:

```
        ORG     100H
START:
        ...
        END     START
```

Output formats for the Microsoft 8086 Linker

RunFile

The Microsoft Linker outputs 'run' files in a relocatable format, suitable for quick loading into memory and relocation. Run files consist of several parts:

- Fixed length header
- Relocation table
- Memory image of resident program

A run file is loaded as follows:

- It is read into memory at any 16 byte (paragraph) boundary
- Relocation is applied to all words described by the relocation table

The resulting relocated program is then executable. Typically, programs in the PL/M small model of computation for the 8086 have little or no relocation; programs in the medium or large model have relocation for long calls, jumps, static long pointers, etc.

The following is a detailed description of the format of a run file, given as an annotated C structure declaration.

```
struct runType {
        short   wSignature;     /* must contain 4D5A hex */
        short   cbLastp;        /* number of bytes contained in last page; this
                                   is useful in reading overlays */
        short   cpnRes;         /* number of 512 byte pages of memory needed to
                                   load the resident and the run file header */
        short   irleMax;        /* number of relocation entries in the table */
        short   cparDirectory;  /* number of paragraphs in run file header */
        short   cparMinAlloc;   /* minimum number of 16 byte paragraphs
                                   required above the end of the loaded
                                   program */
                                /* ignored for Version 1.x of MSDOS, will be
                                   supported in 2.x.
        short   cparMaxAlloc;   /* maximum number of 16 byte paragraphs
                                   required above the end of the loaded
                                   program. 0FFFFh means that the program is
                                   located as low as possible in memory */
        short   saStack;        /* initial value to be loaded into SS before
                                   starting program execution.  This must be
                                   adjusted by relocation */
        short   raStackInit;    /* initial value to be loaded into SP before
                                   starting program execution */
        short   wchksum;        /* negative of the sum of all of the words in
                                   the run file. */
        short   raStart;        /* initial value to be loaded into IP before
                                   starting program execution */
        short   saStart;        /* initial value to be loaded into CS before
                                   starting program execution. This must be
                                   adjusted by relocation */
        short   rbrgrle;        /* relative byte offset from beginning of run
                                   file to the relocation table */
        short   iov;            /* number of the overlay as generated by
                                   LINK-86. The resident part of a program will
                                   have iov = 0 */
};
```

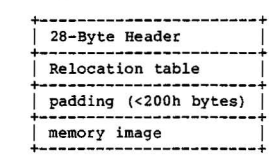
The relocation table follows the fixed portion of the run file header and contains irleMax entries of type rleType, defined by:

```
struct rleType {
        SHORT   ra;
        SHORT   sa;
};
```

Taken together, the ra and sa fields are an 8086 long pointer to a word in the run file to which the relocation factor is to be added. The relocation factor is expressed as the physical address of the first byte of the resident divided by 16. Note that the sa portion of an rle must first be relocated by the relocation factor before it in turn points to the actual word requiring relocation. For overlays, the rle is a long pointer FROM THE BEGINNING OF THE RESIDENT into the overlay area.

The resident begins at the first 512 byte boundary following the end of the relocation table.

The layout of the runFile is:



CONCATENATION WITH THE COPY COMMAND

MS-DOS 1.2 has a variation of the COPY command will allows file concatenation while copying. Concatenation is invoked by simply listing any number of source files to the copy operation, separated by "+". For example,

```
COPY  A.XYZ + B.COM+B:C.TXT BIGFILE.CRP
```

This command will create a new file on the default drive called BIGFILE.CRP, and will put in it the concatenation of A.XYZ, B.COM, and B:C.TXT.

The concatenation operation is normally carried out in text (or ASCII) mode, meaning a Ctrl-Z (1A hex) in the file is interpreted as end-of-file mark. To combine binary files, this may be overridden with the /B switch, which will force the command to use physical end-of-file (i.e., the file length seen in the DIR command). For example,

```
COPY/B A.COM+B.COM
```

Also in this example, no result file name was given. In this case, COPY will seek to the end of A.COM and append B.COM to it, leaving the result called A.COM.

ASCII and binary files may be arbitrarily combined by using /B on binary files and /A on ASCII files. The first example might have intended

```
COPY  A.XYZ + B.COM/B+B:C.TXT/A BIGFILE.CRP
```

A switch (/A or /B) takes effect on the file it is placed after and applies to all subsequent files until another switch is found. Thus the /A after B:C.TXT was necessary.

/A or /B on the destination file determines whether or not a Ctrl-Z will be placed on the end of the file. (Source files read while /A is in effect have Ctrl-Z stipped off. If /A is in effect when the file is written, a single Ctrl-Z will be put back.) Thus an additional Ctrl-Z would be appended with a command such as

```
COPY  A.ASM/B B.ASM/A
```

since the /B on the first file prevents the Ctrl-Z from being stripped, and the /A on the second puts one on. The primary practical application may be the reverse, where a Ctrl-Z is stripped from the file. For example,

```
COPY  PROG.COM/B + ERRMSG.TXT/A  NEWPROG.COM/B
```

It is assumed here that ERRMSG.TXT was generated by an editor, but is actually considered constant data (error messages) by the program it is being appended to. Since the result is a COM file, a Ctrl-Z at the end is not needed.

Even when NOT concatenating files, the /A and /B switches are still processed. When not concatenating, the copy command defaults to binary copy; by using the /A switch, the result file may be truncated at the first end-of-file mark:

```
COPY  A.TXT/A B.TXT
```

B.TXT may be shorter than A.TXT if A.TXT contained a Ctrl-Z before the last character. B.TXT will have exactly one Ctrl-Z, the last character of the file.

Concatenation with ambiguous file names is allowed, and the COPY command normally "does what you want". To combine several files, specified with an ambiguous name, into one file, use a command like

```
COPY  *.LST  COMBIN.PRN
```

All files matching *.LST will combined into one file call COMBIN.PRN. Another type of task is performing several individual concatenations:

```
COPY  *.LST+*.REF *.PRN
```

In this example, for each file matching *.LST found, it will be combined with the corresponding .REF file, with the result given the same name but extension .PRN. Thus FILE1.LST will be combined with FILE1.REF to form FILE1.PRN, then XYZ.LST with XYZ.REF to form XYZ.PRN, and so on. The command

```
COPY *.LST+*.REF COMBIN.PRN
```

will combine all files matchin *.LST, then all files matching *.REF, into one file call COMBIN.PRN.

It is easy to enter a COPY command with concatenation where one of the source files is the same as the destination, yet this often cannot be detected. For example,

```
COPY  *.LST  ALL.LST
```

is an error if ALL.LST already exists. This will not be detected, however, until it is ALL.LST's turn to appended; at this point it could already have been destroyed.

COPY will handle this problem like this: As each input file is found, its name will be compared with the destination. If they are the same, that one input file will be skipped, and the message "Content of destination lost before copy" will be printed. Further concatenation will proceed normally. This does allow "summing" files, with a command like

```
COPY  ALL.LST + *.LST
```

This command will append all *.LST files, except ALL.LST itself, to ALL.LST. The error message will be suppressed in this case, since this is produced by a true physical append to ALL.LST.