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 unit 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 sector. 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.

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.

An example of code which performs this task is given:

```
        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 sector 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 number of 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.

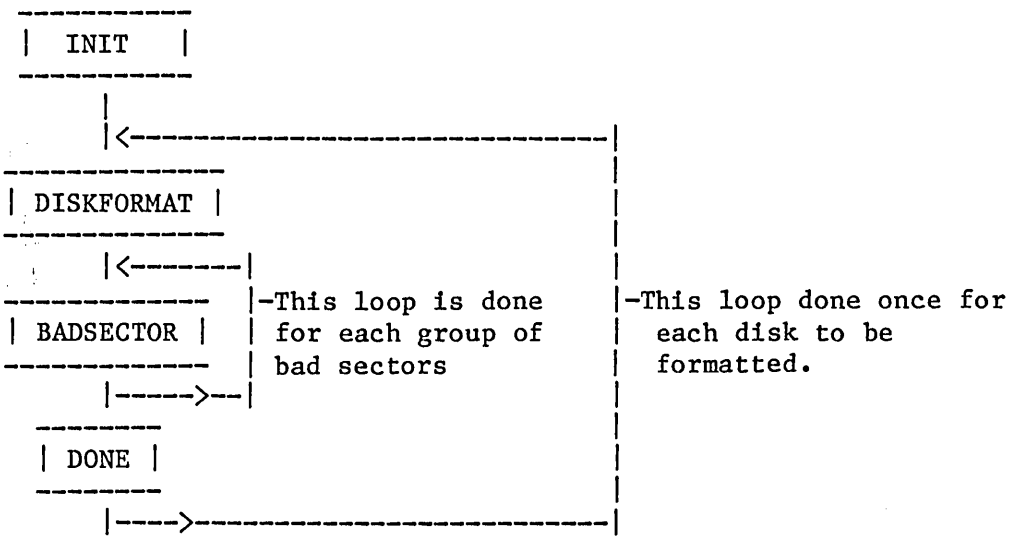
FORMAT - formats a new disk, clears the FAT and DIRECTORY and optionally copies the SYSTEM and COMMAND.COM to this new disk.

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 INIT.OBJ, then the following linker command will do:

```
LINK FORMAT+INIT;
```

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

        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
```

RENDIR.COM

RENDIR allows you to rename directories. The syntax is:

```
RENDIR  oldname  newname
```

The directory being renamed must be in the current directory--full path names are not allowed. Wildcard characters "*" and "?" are allowed in the oldname to rename more that one directory at a time. Wildcard characters in the newname take the characters from the corresponding positions of the original name. (Use of wildcards is identical to the normal REN command).

FUNKEY.EXE

FUNKEY allows you to assign each of the MS-DOS template editing actions to a function key on your terminal or to any desired escape sequence. Each editing action action is initiated by a two-character sequence: the "lead-in" character followed by a character defining which editing action you want. The lead-in character must be the same for all editing actions, and is usually the ASCII ESCAPE code, 1B hexadecimal (27 decimal), although it need not be. The second character may be anything.

Many terminals provide function keys which produce just such a two-character sequence. FUNKEY allows you to select an editing action, then simply type the function key you wish to perform that action. If you do not have enough suitable function keys to assign all editing actions, you can enter a two-character sequence that you will use (remembering that the first character must be the same for all editing actions).

FUNKEY requires you to have a system disk in the default drive. It will read the current assignment of the editing keys from the hidden file MSDOS.SYS on this disk. When you exit FUNKEY, it will update the currently running MS-DOS in memory with the new editing key assignments. It will also prompt you to see if you would like this change put back on the disk. If you exit FUNKEY with Control-C, no changes will be made to disk or memory.

BACKUP is included to simplify the day-to-day keeping of file backup copies. It performs an "incremental" backup--copying only those files which have been changed since the last backup operation. Then by keeping on disk an alphabetical listing of all files, the backup disk for any file can be located quickly by its volume (disk) identification.

BACKUP runs in three phases, and may be stopped with an appropriate switch after any phase. The first phase is to produce an index of disk contents, which requires traversing the entire directory tree. The result of this phase is the file INDEX.NEW in the root directory of the disk being backed up. The file will contain, for each directory on the disk: 1) The path name of the directory; 2) All sub-directories of this directory, listed in alphabetical order, each name indented with two blanks; 3) All files within this directory, listed alphabetically with date, time, and size, indented with one blank. Following the listing for any directory are similar listings for each of its sub-directories.

This first phase can be useful by itself, since it produces an alphabetical listing of the entire disk contents organized by directory. To stop BACKUP at this point, use the /I switch (for Index). For example,

```
BACKUP  /I
```

produces the index file INDEX.NEW for and on the default drive, while

```
BACKUP C: /I
```

makes an index for drive C, leaving it on drive C.

Phase two of BACKUP is to determine which files in the index must be backed up. For speed, this phase actually runs concurrently with phase one if it has not been disabled with the /I switch. BACKUP assumes there is a file named INDEX (no extension) in the root directory of the disk being backed up, which has the index of the disk when it was backed up last. Any file listed in INDEX.NEW which does not match in path name, date, time, and size with a file listed in INDEX will be marked for backup. This marking consists of an asterisk ("*") replacing the blank that is normally in front of the file name. Any file which does match have will have the volume ID of its backup copy listed in INDEX.NEW. If the file INDEX is not found, all files will be marked for backup.

BACKUP may be stopped at the end of this phase by specifying the /N switch (No backup). This allows seeing what files would be backed up if the copying were allowed to take place.

Phase three of BACKUP is to actually perform the copying operations. The file INDEX.NEW will be read back in so that those files marked with the "*" can be backed up. As each file is copied, INDEX.NEW is updated by deleting the "*" and adding the volume ID of the backup disk. If a backup disk fills up, BACKUP will prompt for another. Every disk used for backup MUST have a volume ID so that the index will know where to find the backed up file. No file can be split across disks.

When performing a complete backup, BACKUP uses two drive specifiers: the first for the disk to be backed up (source), the second for the disk on which to make the copies (destination). The destination drive may be omitted if it is

the default drive. For example,

```
BACKUP C: B:
```

backs up drive C onto drive B, while

```
BACKUP B:
```

backs up drive B onto the default drive.

Files are copied into the root directory of the backup disk. This means that if two or more files in different directories have the same name, then as each is copied, it will overwrite the previous one. Thus only the last file of any given name is backed up. This is good if the same file has been copied into more than one directory. However, it is very important for different files to have different basic names, i.e., some difference in the 8-character file name or 3-character extension.

When backup copying is complete, the updated file INDEX.NEW is also copied to the last backup disk. The old INDEX is deleted, and INDEX.NEW is renamed to replace it.

SHIPZONE.COM

SHIPZONE moves the head of the hard disk to the shipping zone, which is cylinder 522 on the MiniScribe 4020. This procedure is ESSENTIAL before a hard disk equipped Gazelle is shipped or moved a significant distance. Failure to do so may damage the drive and will invalidate your warranty. SHIPZONE halts the computer when it is done to be sure no further activity will move the disk heads off the shipping zone. The computer should be turned off after running SHIPZONE.

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.

MEMTEST

STARTING THE TEST

This program tests either the 16K "8/16 RAM" or the 64K RAM from Seattle Computer Products. It can test up to sixteen 16K or 64K boards in any mix. The test prompts the user for information on up to sixteen boards to test.

ADDRESS - enter the beginning address (in hexadecimal) of the board. If you don't know the addresses of your boards, you can use your memory board manuals to interpret the settings of the address switches on the boards.

SIZE - enter either "16" or "64" as appropriate for the board.

DELAY - enter a delay time (0 to 9, A to F seconds) to be used between the write and read passes of the Memory Chip_Test. The delay is designed to find errors in static memory chips which act "dynamic" and forget data shortly after it is written.

There are some restrictions on the addresses of boards which can be tested. MEMTEST is a regular MS-DOS program and runs wherever MS-DOS loads it. Therefore, the test will not allow you to test any memory in which MS-DOS or MEMTEST are residing. To test all the memory in a system, the bottom 32K will have to be swapped with some memory at a higher address. The addresses must also be on a sixteen-byte boundary (i.e. the last digit of the address must be zero).

If less than sixteen boards are to be tested, just enter RETURN when prompted for the address of the next board to start the test. If sixteen boards are tested, the test will start automatically when information for all sixteen boards is entered.

TESTS PERFORMED

Read Only Test - The purpose of this test is to check the data path from the memory chip through the board's output buffers to the bus. It also checks the enabling circuits of the board which allow a memory read operation to take place.

The test operates by assuming the memory board contains random data when the test begins. Each block is scanned to see if any data bits are always high or always low. If the condition is found, an error message is displayed. In case the board happened to have bits all high or low, a word of all zeros and a word of all ones is written to the board to see if both ones and zeros can be read from each bit.

Data Line Test - This test checks the data path into the board from the bus, through the input buffers, to the memory chips, then back out to the bus as does the Read Only Test. The write circuitry is also checked.

This test and the Read Only Test complement one another. The Read Only Test checks the data path out of the board, this one checks the data path entering the board.

The test attempts to write and read back every combination of sixteen bits

into selected locations. If no location is found that passes this simple test, a check is made for data bits that are always high, always low, or shorted together.

Address Line Test - This test checks the address path from the S-100 connector, through the address buffers, to the memory chips.

The address lines are tested by writing a pattern to each memory location which is derived from that locations's address. This makes the test sensitive to addressing problems such as shorted or open address lines, and allows identification of the line(s) with a fault.

Memory Chip Test - The purpose of this test is to detect and isolate defective or marginal memory chips on the board under test.

The test is performed by writing a test pattern into memory and then checking it immediately. After the pattern has been written to the entire board, the program waits for the user specified delay (from 0 to 15 seconds) and then checks the entire board again. The pattern is then "rotated" and the test is performed for a total of 17 times. Then, the pattern is complemented and the test is repeated. After this, a pattern of either all zeros or all ones is written to every location of the board. Next time the Memory Chip Test checks the board, the first thing it does is check this pattern for bits that have changed. This test is intended to find "soft" errors caused by alpha particles.

TEST SEQUENCE

For each of the boards to be tested, the Read Only Test, Data Line Test, and Address Line Test are performed. After these tests have been performed on all boards, the Memory Chip Test is done to each board. The test then loops indefinitely doing the Memory Chip Test over and over until stopped.

STOPPING THE TEST and READING TEST RESULTS

Any errors which occur while the test is running are stored so they can be printed later. If control-C is typed anytime during the test, the entire history of successes or failures for each board is printed and control is returned to the MS-DOS. Anytime during the Memory Chip Test, MEMTEST will also accept ESCAPE which will print the success/failure history and resume the test.

INTERPRETATION OF TEST RESULTS

MEMTEST indicates errors in terms of "bits" and "blocks" which correspond to the bits and blocks of the SCP 16K "8/16 RAM" and 64K Static RAM boards as defined in the Trouble-Shooting sections of these board's manuals.

These boards each have two blocks of memory chips. For the 16K "8/16 RAM" the blocks are 8K bytes (or 4K words) long, and for the 64K Static RAM the blocks are 32K bytes (or 16K words) long. All the blocks are sixteen bits wide. The block number defines the tens digit of a memory chip's IC number. The bit number defines the ones digit of a memory chip's IC number. For example, if an error is indicated in bit "E" of block "2", U2E would be the bad chip.

When interpreting test results, it must be remembered that the memory test is a sequential test consisting of several sub-tests. The significance of this fact is that the validity of sub-tests late in the sequence is dependent upon

successful earlier sub-tests. For this reason, the earliest sub-test in which a fault is detected can be the most significant. Examples:

If the board fails the Read Only Test, the results of the rest of the tests can't be believed since they depend on the data path from the memory chips out to the data bus to work.

If the board fails the Address Line Test, the Memory Chip Test will not produce reliable results because the address problems will usually cause all bits of all blocks to fail the Memory Chip Test.