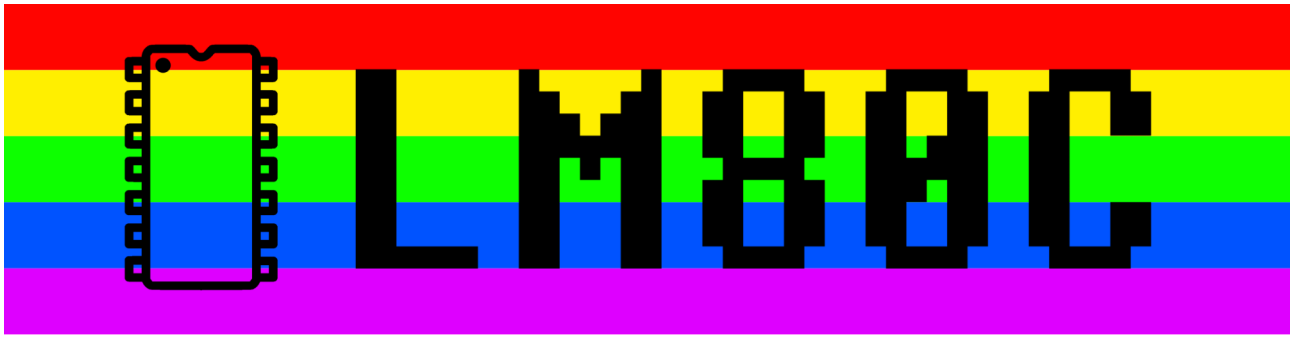

LM80C DOS

DISK OPERATING SYSTEM

LM80C DOS
DISK OPERATING SYSTEM
REFERENCE MANUAL

This release covers the
LM80C DOS Disk Operating System

Latest revision on 21/04/2021

Copyright notices:

Index

1. The LM80C DOS

The **LM80C DOS** is a Disk Operating System integrated into the firmware of the LM80C 64K Color Computer that serves as an interface to make input/output operations with a mass storage device. The devices used as mass storage are common Compact Flash (CF) cards (from now on, “disk” will be used in this document as a synonym for CF card). CF cards have been chosen because they can operate in 8-bit mode: this is important when talking about interfacing them with the Z80 CPU since this microprocessor has an 8-bit data bus that can be directly connected to such memories. Moreover, they are relatively cheap and large enough to store an obscenely amount of files if compared to floppy disks of the time. CF cards can be driven in both IDE and LBA modes: the latter is the one used on LM80C Color Computers. In LBA (Logical Block Addressing) each sector of the card can be addressed by setting up its sector number through the cards’ registers.

1.1 How the disk is formatted

The LM80C DOS formats (initializes) a disk by dividing its space in 3 main areas:

1. Master Sector (MS)
2. Directory (DIR)
3. Data Area (DA)

1.2 The Master Sector

The Master Sector is what in other DOSs is often called the MSB, or Master Boot Sector. It is always the sector #0 of the disk and contains specific details on how the disk was formatted and info of the disk itself like the name, the geometry of the disk, the starting addresses of the directory and DA. The MS contains the following data:

```
$000-$008:  "LM80C-DOS"
$009:       $00
$00A-$00D:  "A.BC" (4 byte) ← version of DOS used to format the disk (i.e.:
            1.06)
$00E:       $00
$00F-$012:  disk size in sectors (4 bytes): value recovered from bytes $00E-
            $011 of the CF card ID*
$013-$014:  number of cylinders (2 bytes): value recovered from bytes $002-
            $003 of the CF card ID*
$015-$016:  sectors per track (2 bytes): value recovered from bytes $00C-
            $00D of the CF card ID*
$017-$018:  number of heads (2 bytes): value recovered from bytes $006-$007
            of the CF card ID*
$019-$01A:  number of allowed files (2 bytes)
$01B-$01C:  starting sector of the directory (2 bytes) - default $0001
$01D-$01E:  starting sector of the Data Area (2 bytes)
$01F:       $00
$020-$02F:  disk name (16 bytes) – unused bytes are filled up with $20
            (ASCII 32, space)**
$030-$033:  disk ID (4 bytes) – one letter, one number, one letter, and one
            number (chars "A" to "Z" and "0" to "9", ex.: "B1Y7")
$034-$1FD:  $00
```

```
$1FE-$1FF:  "80" ($38 $30) ← control signature
```

*CF card ID: the “Identify Drive” is a command accepted by the CF card that returns some useful details about the card itself.

**disk name: allowed chars are letters from “A” to “Z”, numbers from “0” to “9”, “space” and “-” (minus symbol).

IMPORTANT: words (2 bytes) and double words (4 bytes) are stored as follow. For single words, the Little Endian format is used: this means that the first byte contains the less significant byte (LSB) while the second byte contains the most significant byte (MSB). For double words, the first couple of bytes contains the MSW (most significant word), stored in Little Endian format, while the second couple contains the LSW (less significant word), also stored in Little Endian format.

Here is an example of a Master Sector from a 256 MB CF card:

```
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------------------
$000  4C 4D 38 30 43 20 44 4F 53 00 31 2E 30 36 00 07 LM80C DOS.1.06..
      |-------- DOS name  -----| -- |DOS vers.| -- |-
$010  00 00 A8 D4 03 20 00 10 00 50 0F 00 01 F6 00 00 ..(T. ...P...v..
      sectors| |cyl| |sct| |hds| |fil| |DIR| |DAT| --
$020  54 45 53 54 44 49 53 4B 20 20 20 20 20 20 20 20 TESTDISK
      |--------------   disk  name   ---------------|
$030  54 33 45 37 00 00 00 00 00 00 00 00 00 00 00 00 T3E7............
      |-- ID  --|
$040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
....
....
$0F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 38 30 ..............80
```

By analyzing such data we can get some useful information. We can see that the DOS version is 1.06, that “TESTDISK” is the disk name, that the directory starts at sector $0001 and that the DA starts at sector $0000.00F6. The total amount of sectors of the disk are $0007.A800 (501,760). In fact, the disk geometry can be found using this formula:

```
cylinders × sectors per cylinder × heads
```

that gives the following results:

```
$03D4 × $20 × $10 = $7A800 = 501,760
```

To know the real size of a CF card, we multiply this value by the size of a sector. This value is fixed for any CF card, and is equal to 512 bytes. So, translated in numbers:

```
501,760 × 512 = 256,901,120 bytes = 245 MB
```

This is the formatted space over the nominal size (256 MB): keep in mind that no CF card makes available the whole capacity.

1.3 The Directory

The **Directory** is the disk area that is intended to keep the details about the files store into the disk. Every file descriptor is called an **entry**. Entries are also the file indexes themselves, since each entry in the directory corresponds to a file. Files are stored in **blocks**: each block of the disk is 64KB wide, and each block can store only one file. So, the 1st entry in the directory corresponds to the 1st file on the disk; the 2nd entry corresponds to the 2nd file; and so on. The size of the directory is calculated when the disk is formatted and it's based on the size of the disk itself. Since a sector of the disk is 512 bytes wide, and since each entry in the directory occupies 32 bytes, we can easily calculate that each sector of the directory contains 16 entries (512/32=16). The maximum allowed number of files that can be managed by LM80C DOS is 65,536, no matter the size of the CF card used is. So, the maximum number of sectors occupied by the directory is:

```
65,536 / 16 = 4,096
```

To calculate the size of the directory we must first find the number of sectors available (see 1.1.1) and then divide this number by the number of sectors per block. We already said that each block is 64KB wide. Given this, we now know that each block contains 128 sectors, because:

```
block size in sectors = block size in bytes / bytes per sector
```

So, we have:

```
65,536 / 512 = 128
```

The maximum number of entries in a directory are given by:

```
disk size in sectors / sectors per block
```

For our current example with 501,760 sectors, we can find that this disk can contain 3,920 entries because:

```
501,760 / 128 = 3,920
```

1.3.1 A directory entry

Each entry in the directory is composed by 32 bytes, each byte carries specific info, as in the table below:

```
$00-$0F: file name (16 bytes) - allowed chars are letters from “A” to “Z”,
         numbers from “0” to “9”, space and “-” (minus). Extensions are not
         supported.
$10:     type of file – BAS: $82; BIN: $81; $82: SEQ
$11:     file attributes (at the moment, NOT supported)
$12-$13: entry number (2 bytes)
$14-$17: number of 1st sector of the corresponding block (4 bytes)
$18-$19: size in bytes (2 bytes)
$1A:     size in sectors (1 byte)
$1B-$1C: starting address of the file in the computer memory
```

```
$1D-$1F:$00 (reserved for future uses)
```

The first char of the file name can also have a couple of values with specific meanings:

- a value of $00 means that the entry is empty (never used before) and free for use;
- a value of $7F means that the entry has been deleted and can be re-used.

Below is an example of an entry of the directory of our sample disk:

```
4D 41 52 49 4F 20 20 20 20 20 20 20 20 20 20 MARIO
 |  ----------------- file name -------------|
80 00 00 00 00 00 F6 00 37 00 01 07 5E 00 00 00 ........7.......
tp at |ent| |-sector -| |siz| st |adr| --------
```

By looking at the entry's data we can see that:

- the file name is "MARIO" (look at the padding spaces, $20, added to fill up the file name space);
- this is a BASIC file (type $80);
- it has no attributes ($00);
- its entry number is $0000;
- the block that stores its data begins at sector $00F6;
- its size is $0037 bytes, corresponding to 1 sector;
- the file originally was located in RAM starting at $5E07.

1.4 File types

The LM80C DOS treats files regarding of their type. There is no support for "extensions", as intended in other DOSs, i.e. no portions of the file name can be introduced by a "." (full stop) to set the type of file: instead, the type is written directly by DOS into the corresponding file entry in the directory. When listing files of a disk, the DOS will always print the file type after the file name.

Currently, the following file types are supported:

BAS: BASIC programs ($80)

BIN: binary files ($81)

SEQ: sequential files ($82)

???: unknown (a disk error may have been occurred and ruined the file entry)

1.4.1 BASIC files

A BASIC file (marked as “BAS” when listed) is loaded in memory ignoring the address stored into it but looking at the BASIC pointers of the computer. This is important because a file saved for a different version of firmware could be loaded in a BASIC area that is *not* correctly aligned with the pointers of the area of the computer used to save the file. This is the reason why the DOS version of the disk is always checked before to operate on it, so that a disk can not be used if the DOS version used to format such disk doesn’t match the DOS being executed. The same control can not be executed on BASIC files being loaded so it is the user that has to avoid loading a program saved with a different firmware.

When a “BAS” file is saved, the DOS makes also a copy on the disk of a specific portion of memory, so that when the files is loaded up again, it will result as if it was just typed in by the user. The area that is stored goes from `$PROGND` to `($PROGND)`, so that several BASIC pointers are saved together with the program itself. What do those values stand for? Like in assembly, `$PROGND`, stands for the address of that memory cell, while `($PROGND)` stands for the pointer stored into that cell: this pointer is a word and points at the last memory cell occupied by the BASIC program. When the BASIC program is loaded into memory, its bytes will be restored from such address, recovering the exact status of the BASIC environment when the program was saved.

1.4.2 Binary files

Different is the case of Binary (“BIN”) files. Binary files are particular files which contain a raw copy of a portion of memory. In this case, it is mandatory to force the DOS loading the file respecting the original address of the data (argument “,1” after the `LOAD` statement). It is important that the user knows exactly where to load a “BIN” file into memory because its contents may be copied over a memory area occupied by BASIC or DOS routines, altering the functioning of the computer and, in the worst case, leading to the necessity of an hard reset to restore the original firmware.

1.4.3 Sequential files

Sequential files (“SEQ” type) are files whose data are stored in chronological order as they are saved into them This means that in order to locate a data, such files must be read starting at the beginning of the file and so on, until the position of the required data is reached. A sequential file can be increased in size by adding additional data at the end of the file (“appending”): when opening a file already present into the directory, the DOS will position the data pointer at the end of such file to add other data, otherwise it will create a new empty file.

During the data inserting, the DOS keeps the sector that is reflecting the portion of file where the DOS is inserting data in. When the user inserts into/read from the buffer, the data pointer increments itself until the end of the buffer. When writing, as soon as the buffer is full, the DOS will save the current sector onto the disk, then will create a new empty buffer. When reading, after the DOS has reached the end of the buffer, it will load the next sector, if the file keeps more data, or raises an EOF (End-Of-File) error otherwise.

1.5 File entries management

Since the LM80C DOS doesn't make use of file allocation tables, but instead it makes use of a simpler mechanism where each directory entry is also the pointer to the file block, the DOS must scan the entire directory to know which entries are free and which aren't. But, since the disk is very fast, scanning a directory with thousands of entries only require few seconds. Every time a new file is created, the DOS scans the directory looking for the first usable entry, halting its course at the first match. When listing files, instead, the DOS scans the whole directory to seek for entries.

When a file has been deleted, the user can choose 2 different deleting methods: a quick erase, and a full erase. The first one just marks the file to be deleted by writing the special code $7F in the first char of its name, leaving its data on the disk unchanged. The user can, by using the DISK function "U" or by manually changing this value in the sector of the entry, recover the file easily. The second method is more secure but it can not be reverted: not only the entry is completely wiped out but also the sectors occupied by the file data are erased, deleting their bytes permanently.

Each block is assigned to one, and only one, file. Sectors not used in a block can not be assigned to other blocks and remain unused. No more than 65,536 blocks (and even files) can be stored in a disk. Very big disks won't be utilized in their whole capacity. This can appear as a big waste of space but since the Z80 is an old CPU with limited capabilities, this seems to be a good compromise between resource allocated to address the disk and computer memory utilized by DOS.

1.6 The Data Area

The area used to store the file data is called Data Area and it is divided into blocks of a fixed size, 128 sectors corresponding to 64KB (65,536 byte). This is also the maximum size that a file can have.

The DA follows the directory and covers the rest of the disk space. The first sector of the DA can be calculated using the following formula:

```
1 + (number of entries / entries per sector)
```

Using our sample disk with 3,920 entries, the DA begins at:

```
1 + (3,920 / 16) = 246
```

This means that the directory, always starting at sector #1, will occupy 246 sectors, from #1 to #245. The DA will then begins at sector #246.

Every byte of a sector is used to store data. The mechanism to keep trace of where to stop loading data from disk is based on the file size stored in its entry. If a file needs more than 1 sector to store its data, to know how many bytes are in the last sector a simple subtraction is done. To know the address of the last byte of the last sector the following formula is used:

```
xxx = (512 * file size in sectors) – (file size in bytes)
```

These values are recovered from the file entry.

Let's say that a file is 13,512 bytes in size, occupying 27 sectors. Using the above formula, we can find the following:

```
512 * 27 = 13824 -
           13512=
           -----
             312
```

So, the last byte is given by "xxx" – 1. Using the example above:

```
312 - 1 = 311 ($137)
```

This means that bytes from $000 to $137 will contain regular data, while bytes from $138 on will contain garbage ignored by the DOS.

1.7 The DOS buffers

While doing its jobs, the LM80C DOS exchanges data from the disk to the computer and vice-versa. Since a single sector is 512 bytes wide, the DOS needs a place where to temporary park this stream of bytes because the CPU isn't able to deal with it. To accomplish this, 4 temporary memories called "*buffers*" are used to store data from/to the disk.

The first buffer is the I/O Buffer (Input/Output Buffer). Since it has to store an entire sector, its size is 512 bytes. The I/O buffer is used to create the 512 bytes that will be stored on the disk, or to park the bytes of a sector when read from the disk.

The second buffer is placed just below the previous one and it's 32 bytes wide. This is the DOS buffer, and it's used primary to, but not limited to, store an entire entry of the directory during loading & saving operations.

The third buffer is 36 bytes wide and it's used as a support buffer to store temporary data used by BIOS routines to read/write data from/to the disk. It's located in the BASIC workspace.

The last buffer is 27 bytes wide and it's used by DOS to manage sequential files. This area is located just over the I/O buffer.

1.8 DOS memory occupation

After the boot, and if it's enabled, the DOS is copied (and will be reside) in the higher part of the RAM. The first portion of the memory contains the DOS, then the BIOS routines, then the I/O buffers, and finally the DOS jump table.

The beginning of DOS is at $EE70. The DOS contains the routines to load, save, erase, rename, and list files from/to the disk and the BASIC bindings to execute LOAD, SAVE, ERASE, FILES, and DISK statements.

After the DOS, from location BIOSSTART ($FCCB in current firmware 1.16), there is the BIOS (Basic Input Output System), a group of routines used to access the Compact Flash card so that the DOS can access it without the need to know the hardware specs of the system under its software layer. The BIOS ends just before the space occupied by the I/O & DOS buffers.

After the BIOS, the first buffer is the DOS buffer. It resides from $FDA0 up to $FDBF, and it's 32 bytes wide. Then, the I/O buffer occupies the space between $FDC0 and $FFBF, 512 bytes. After it, there is the sequential file buffer from $FFC6 to $FFE0 (27 bytes wide)

At the very top of the RAM there is the DOS jump table. This table contains the jumps to the current BASIC routines for the DOS commands. When the DOS is not enabled, the table entries point to the REM routine so that calling the DOS commands has no effect.

```
+--------+  $FFFF          top of RAM
|        |  $FFFD-$FFFF    jump to FILES command
|        |  $FFFA-$FFFC    jump to SAVE command
|        |  $FFF7-$FFF9    jump to LOAD command
|        |  $FFF4-$FFF6    jump to ERASE command
|        |  $FFF1-$FFF3    jump to DISK command
|        |  $FFEE-$FFE0    jump to OPEN command
|        |  $FFEB-$FFED    jump to CLOSE command
|        |  $FFE8-$FFEA    jump to GET function
|        |  $FFE5-$FFE7    jump to PUT command
|        |  $FFE2-$FFE4    jump to EOF function
+--------+  $FFE1-$FFDF    jump to EXIST function
+--------+  $FFDE          top of seq. File buffer
|        |
+--------+  $FDC4          bottom seq. file buffer
|        |  $FFC0-$FFC3    filler
+--------+  $FFBF          top of I/O buffer
|        |
|        |  $FDC0          bottom of I/O buffer
+--------+  $FDBF          top of DOS buffer
|        |
+--------+  $FDA0          bottom of DOS buffer
|        |  $FD9B-$FD9F    filler
+--------+  $FD9A          top of BIOS
|        |
|        |  $FCCB          bottom of BIOS
+--------+  $FCCA          top of DOS
|        |
|        |
|        |
+--------+  $EE68          bottom of DOS
|        |  $EE67          top of string space
|        |  $EE04          bottom of string space (assuming 100 bytes)
+--------+  $EE03          top of stack
|        |
```

2. USEFUL LINKS

Project home page:

https://www.leonardomiliani.com/en/lm80c/

Github repository for source codes and schematics:

https://github.com/leomil72/LM80C

Hackaday page:

https://hackaday.io/project/165246-lm80c-color-computer

LM80C
Color Computer

Enjoy home-brewing computers

Leonardo Miliani