An Inside Look at MS-DOS

The design decisions behind the popular operating system

Tim Paterson
Seattle Computer Products
1114 Industry Dr.
Seattle, WA 98188

The purpose of a personal computer operating system is to provide the user with basic control of the machine. A less obvious function is to furnish the user with a high-level, machine-independent interface for application programs, so that those programs can run on two dissimilar machines, despite the differences in their peripheral hardware. Having designed an 8086 microprocessor card for the S-100 bus and not finding an appropriate disk operating system on the market, Seattle Computer Products set about designing MS-DOS. Today MS-DOS is the most widely used disk operating system for personal computers based on Intel's 8086 and 8088 microprocessors.

MS-DOS Design Criteria

The primary design requirement of MS-DOS was CP/M-80 translation compatibility, meaning that, if an 8080 or Z80 program for CP/M were translated for the 8086 according to Intel's published rules, that program would execute properly under MS-DOS. Making CP/M-80 translation compatibility a requirement served to promote rapid development of 8086 software, which, naturally, Seattle Computer was interested in. There was partial success: those software developers who chose to translate their CP/M-80 programs found that they did indeed run under MS-DOS, often on the first try. Unfortunately, many of the software developers Seattle Computer talked to in the earlier days preferred to simply ignore MS-DOS. Until the IBM Personal Computer was announced, these developers felt that CP/M-86 would be *the* operating system of 8086/8088 computers.

Other concerns crucial to the design of MS-DOS were speed and efficiency. Efficiency primarily means making as much disk space as possible available for storing data by minimizing waste and overhead. The problem of speed was attacked three ways: by minimizing the number of disk transfers, making the needed disk transfers happen as quickly as possible, and reducing the DOS's "compute time," considered overhead by an application program. The entire file structure and disk interface were developed for the greatest speed and efficiency.

The last design requirement was that MS-DOS be written in assembly language. While this characteristic does help meet the need for speed and efficiency, the reason for including it is much more basic. The only 8086 software-development tools available to Seattle Computer at that time were an assembler that ran on the Z80 under CP/M and a monitor/debugger that fit into a 2K-byte EPROM (erasable programmable read-only memory). Both of these tools had been developed in house.

MS-DOS Organization

The core of MS-DOS is a device-independent input/output (I/O) handler, represented on a system disk by the hidden file MSDOS.SYS. It accepts requests from application programs to do high-level I/O, such as sequential or random access of named disk files, or communication with character devices such as the console. The handler processes these requests and converts them to a very low level form that can be handled by the I/O system. Because MSDOS.SYS is hardware independent, it is nearly identical in all MS-DOS versions provided by manufacturers with their equipment. Its relative location in memory is shown in figure 1.

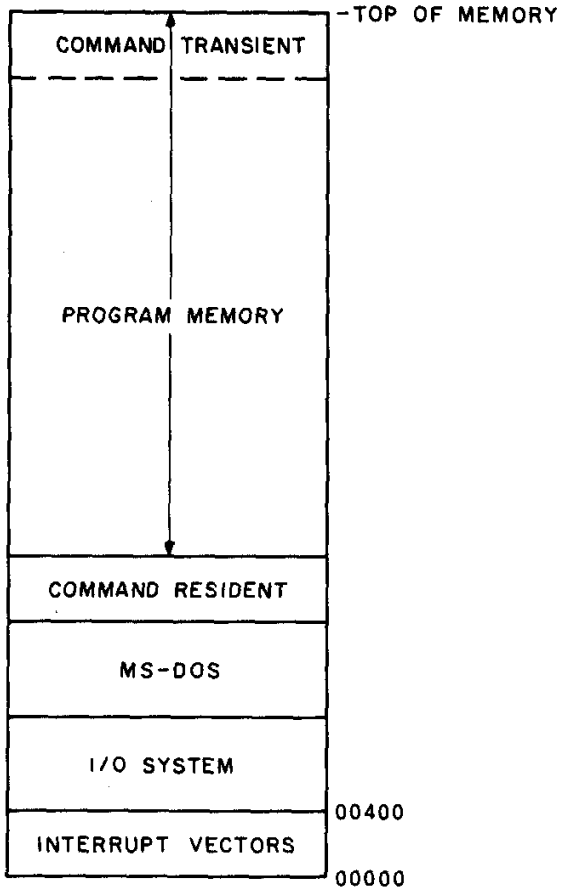


Figure 1: *Map of memory areas as assigned by MS-DOS.*

The I/O system is totally device dependent and is represented on the disk by the hidden file IO.SYS. It is normally written by hardware manufacturers (who know their equipment best, anyway) with the notable

exception of IBM, whose I/O system was written to IBM's specifications by Microsoft. The tasks required of the I/O system, such as outputting a single byte to a character device or reading a contiguous group of physical disk sectors into memory, are as simple as possible.

The command processor furnishes the standard interface between the user and MS-DOS and is contained in the visible file COMMAND.COM. The processor's purpose is to accept commands from the console, figure out what they mean, and execute the correct sequence of functions to get the job done. It is really just an ordinary application program that does its work using only the standard MS-DOS function requests. In fact, it can be replaced by any other program that provides the needed user interface.

There are, however, two special features of the COMMAND file. First, it sets up all basic error trapping for either hard-disk errors or the Control-C abort command. MSDOS.SYS provides no default error handling but simply traps through a vector that must have been previously set. Setting the trap vector and providing a suitable error response is up to COMMAND (or whatever program might be used to replace it).

The second special feature is that COMMAND splits itself into two pieces, called the *resident* and *transient* sections. The resident, which sits just above MS-DOS in low memory, is the essential code and includes error trapping, batch-file processing, and reloading of the transient. The transient interprets user commands; it resides at the high end of memory where it can be overlaid with any applications program (some of which need as much memory as they can get). This feature is of limited value in systems with large main memory, and it need not be imitated by programs used as a replacement for COMMAND.

COMMAND provides both a useful set of built-in commands and the ability to execute program files located on the disk. Any file ending with the extensions .COM, .EXE, or .BAT can be executed by COMMAND simply by typing the first part of the file name (without extension). You can normally enter parameters for these programs on the command line, as with any of the built-in commands. Overall, the effect is to give you a command set that can be extended almost without limit just by adding the command as a program file on the disk.

The three different extensions allowed on program files represent different internal file formats.

- .COM files are pure binary programs that will run in any 8086 memory segment; in order for this to be possible, the program and data would ordinarily have to be entirely in one 64K-byte segment.
- .EXE files include a header with relocation information so that the program may use any number of segments; all intersegment references are adjusted at load time to account for the actual load segment.
- .BAT (batch) files are text files with commands to be executed in sequence by COMMAND.

File Structure

Disks are always divided up into tracks and sectors, as shown in figure 2. To access any particular block of data, the program first moves to the correct track, then has you wait while the spinning disk moves the correct sector under the head.

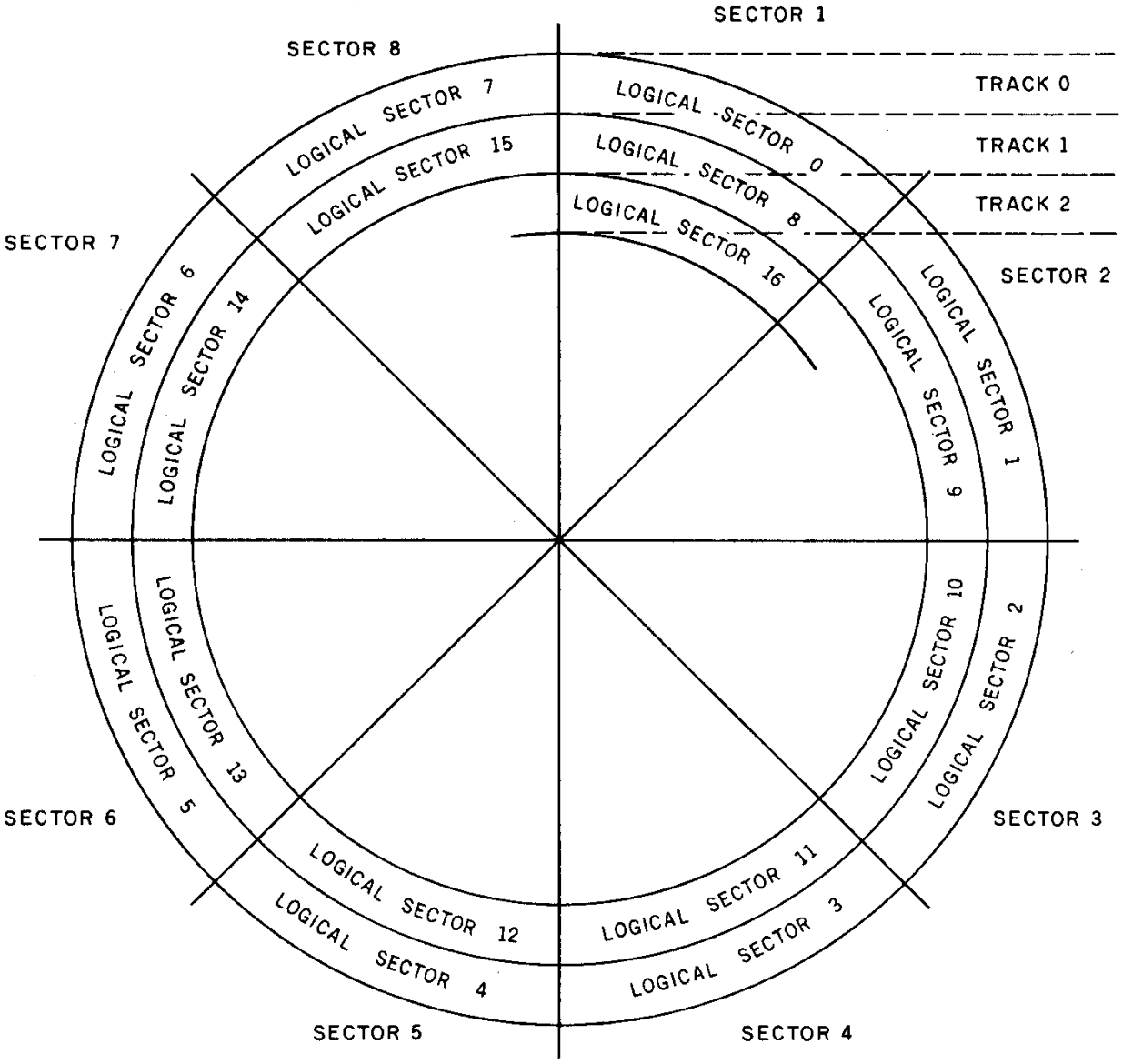


Figure 2: *Placement of disk sectors in IBM Personal Computer (single-sided) format.*

A somewhat more abstract view of disks was taken in developing MS-DOS. MS-DOS views the disk, not in terms of tracks and sectors, but as a continuous array of *n* logical sectors, numbered from 0 to *n* - 1. Figure 2 shows the usual method of numbering the logical sectors. Logical sector 0 is the first sector of the outermost track; the rest of the track (and the next, etc.) is numbered sequentially. Logical sector *n* - 1 is the last sector on the innermost track.

The mapping of logical sectors to physical track and sector is done by the hardware-dependent I/O System and is completely transparent to the MS-DOS file system. Any other method may be used, and

MS-DOS wouldn't know the difference. Having a standard mapping, however, is essential for interchanging disks between computer systems with different peripheral hardware.

As shown in table 1, the MS-DOS file system divides the linear array of logical sectors into four groups. The first of these is the reserved area, whose purpose is to hold the bootstrap loader. Because the loader is usually very simple, only one sector is normally reserved.

Logical Sector Numbers	Use	
0	Reserved for bootstrap loader	
1-6	FAT 1	file allocation tables (FATs)
7-12	FAT 2	
13-29	Directory	
30-2001	Data	

Table 1: *Map of disk areas on an 8-inch single-sided, single-density floppy disk.*

The FAT (file allocation table), a map of how space is distributed among all files on the disk, comes next. Because it is so important, two copies are usually kept side by side. If one copy cannot be read because of a failure in the medium, the second will be used.

The directory follows the FAT. Each file on the disk has one 32-byte entry in the directory, which includes the file name, size, date and time of last write, and special attributes. Each entry also has a pointer to a place in the FAT that tells where to find the data in the file. Figure 3 shows the layout of a directory entry.

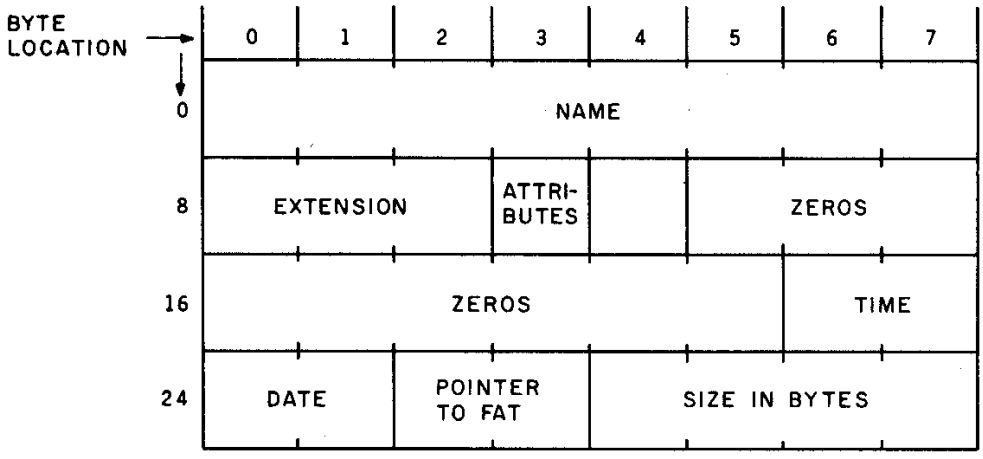


Figure 3: *Arrangement of bytes in disk directory entry.*

The rest of the disk is the data area. It is divided into many small, equal-sized areas called *allocation*

units. Each unit may have 1, 2, 4, 8, 16, 32, 64, or 128 logical sectors, but the number is fixed for a given disk format. Allocation units are numbered sequentially. The numbering starts with 2; the first two numbers, 0 and 1, are reserved. Table 2 shows this numbering system applied to the 8-inch single-density disk format.

Logical Sector Numbers	Allocation Unit Number
30 - 33	2 (first allocation unit)
34 - 37	3
38 - 41	4
42 - 45	5
. . .	. . .
1998 - 2001	494

Table 2: *Allocation unit numbering for the 8-inch single-density format. To compute the logical sector number of the first sector in an allocation unit, you use the following equation: sector number = 4 x allocation unit number + 22.*

The allocation unit is the smallest unit of space MS-DOS can keep track of. The amount of space used on the disk for each file is some whole number of allocation units. Even if the file is only 1 byte long, an entire unit will be dedicated to it.

For example, the standard format for 8-inch single-density disks uses four 128-byte sectors per allocation unit. When a new file is first created, no space is allocated, but an entry is made in the directory. Then when the first byte is written to the file, one allocation unit (four sectors) is assigned to the file from the available free space. As each succeeding byte is written, the size of the file is kept updated to the exact byte, but no more space is allocated until those first four sectors are completely full. Then to write 1 byte more than those four sectors worth, another four-sector allocation unit is taken from free space and assigned to the file.

When writing stops, the last allocation unit will be filled by some random amount of data (figure 4). The unused space in the last allocation unit is wasted and can never be used as long as the file remains unchanged on the disk. This wasted space is called *internal fragmentation*, because it is part of the space allocated to the file but is an unusable fragment. On the average, the last allocation unit (regardless of size) will be half filled and, therefore, half wasted. Because each file wastes an average of one-half an allocation unit, the total amount of space wasted on a disk due to internal fragmentation is the number of files times one-half the allocation unit size.

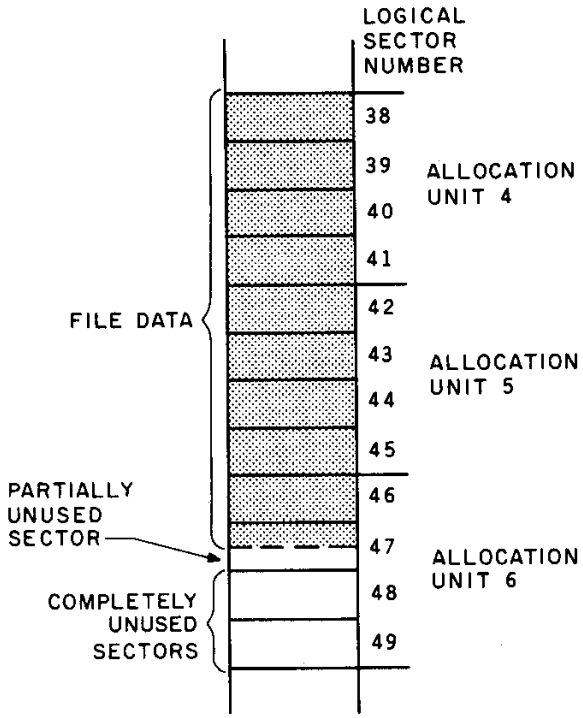


Figure 4: *Assignment of logical sectors to allocation units. Note that, in the file shown, more than two sectors are wasted because they are in an unused part of the last allocation unit.*

The phenomenon called *external fragmentation* occurs when a piece of data space is unallocated yet remains unused because it is too small. This cannot happen in the MS-DOS file system because MS-DOS does not require files to be allocated contiguously. It is, however, present in more primitive systems, such as the UCSD p-System.

It would certainly seem desirable to minimize internal fragmentation by making the allocation unit as small as possible -- always one sector, for example. However, for any given disk size, the smaller the unit, the more there must be. Keeping track of all those units can get to be a problem. Specifically, the amount of space required in the file allocation table would be quite large if there were too many small allocation units. For every unit, 1.5 bytes are required in the FAT; there are normally two FATs on the disk, each of which is rounded up to a whole number of sectors.

Now take a standard 8-inch single-density floppy disk that has 2002 sectors of 128 bytes. To minimize internal fragmentation, choose the smallest possible allocation-unit size of one sector. Two thousand allocation units will require 3000 bytes (24 sectors) per FAT, or 48 sectors for two FATs. If the average file size is 16K bytes (128 sectors), the disk will be full when there are 16 files on it. Waste due to internal fragmentation would be

16 files x 64 bytes per file = 1024 bytes (8 sectors)

Far more space is occupied by the FATs on the disk than is wasted by internal fragmentation!

To provide maximum usable data space on the disk, both internal fragmentation and FAT size must be considered because both consume data area. The standard MS-DOS format for 8-inch single-density disks strikes a balance by using four sectors per allocation unit. Two sectors per unit would have been just as good (assuming a 16K-byte average file size), but there is another factor that always favors smaller FATs and larger allocation units: the entire FAT is kept in main memory at all times.

The file allocation table contains *all* information regarding which allocation units are part of which file. Thus by keeping it in main memory, any file can be accessed either sequentially or randomly without going to disk except for the data access itself. Schemes used in other operating systems (including CP/M and Unix) may require one or more disk reads simply to find out where the data is, particularly with a random access. In an application such as a database inquiry, where frequent random access is the rule, this can easily make a 2 to 1 difference in performance.

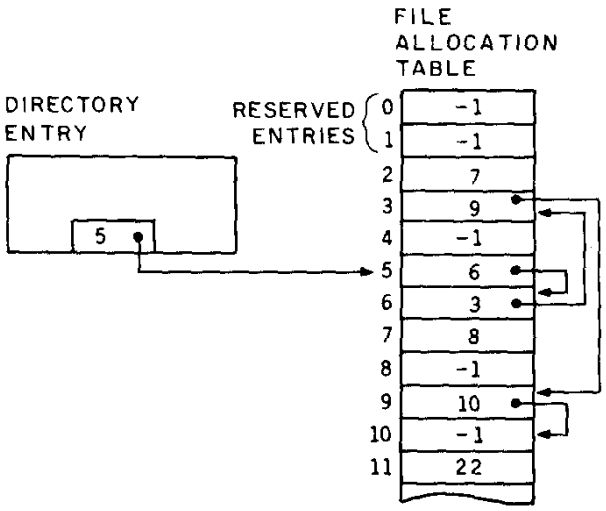
How the FAT Works

The directory entry for each file has one allocation unit number in it: the number of the first unit in the file. If, as in the previous example, an allocation unit consists of four sectors of 128 bytes each, then just by looking at the directory you know where to find the first 512 bytes of the file. If the file is larger than this, you go to the FAT.

The FAT is a one-dimensional array of allocation unit numbers. As with any array, a given element is found with a numeric index. The numbers used as indexes into the FAT are also allocation unit numbers. Think of the FAT as a map, or translation table, that takes an allocation unit number as input and returns a different allocation unit number as output. The input can be any unit that is part of a file; the number returned is the next sequential unit of that file.

Let's look at the example in figure 5a. Suppose that the directory entry for a file specifies allocation unit number 5 as the first of the file. This locates the first four logical sectors (512 bytes). To find the next allocation unit of the file, look at entry 5 in the file allocation table. The 6 there tells you two things: first, the next four logical sectors of the file are in allocation unit number 6; and second, to find the unit after that, look at FAT entry number 6.

(5a)



(5b)

```
FF FF FF 07 90 00 FF 6F 00 03 80 00 FF AF 00 FF 6F 01
```

Figure 5: *Finding data via the directory and the file allocation table. Figure 5a shows how pointers are used to direct the operating system to the sequential parts of a file. The data stored in the sample file allocation table is displayed in hexadecimal in figure 5b.*

This process is repeated as you locate each allocation unit in the file. After number 6 comes number 3, then number 9, then number 10. In each case, the allocation unit number returned by the FAT tells you both where the data is (i.e., in which unit) and where to look in the FAT for the next allocation unit number.

If you followed the example all the way through, you should have noticed that entry 10 in the FAT contains a -1. This, as you might have guessed, is the end-of-file mark. Allocation unit number 10 is last in the file, so its entry contains this special flag. (Another special value in the FAT is 0, which marks a free allocation unit.)

Now you know that this file occupies five allocation units, numbers 5, 6, 3, 9, and 10 in order. The file could be extended by finding any free unit, say 27, putting its number in entry 10 (where the -1 is now), and marking it with the -1 for end of file. That is, entry 10 in the FAT will contain 27, and entry 27 will contain -1. This demonstrates how you can extend any file at will and that you can use any free space on the disk when needed, without regard to its physical location.

If you ever want to look at a real FAT, there's one more thing you'll need to know. Each FAT entry is 12 bits (1.5 bytes) long. These entries are packed together, so two of them fit in 3 bytes. From a programming viewpoint, you would look up an entry in the FAT this way: Multiply the entry number by 1½, truncating it to an integer if necessary. Fetch the 16-bit word at that location in the FAT. If the original entry number was odd (so that truncation was necessary in the first step), shift the word right 4

bits; the lowest 12 bits of the word is the contents of the FAT entry. Reading a FAT from a hexadecimal dump isn't nearly as simple! Figure 5b shows the hexadecimal version of the sample FAT I've been using.

File System in Action

To put all this in perspective, you need to look at how MS-DOS handles a file-transfer request from an application program. With MS-DOS, application programs treat files as if they were divided into logical records. The size of the logical record is entirely dependent on the application and may range from 1 byte to 65,535 bytes. It is not a permanent feature of a file but in fact may vary from one file-transfer request to the next. The logical record size currently being used is passed to MS-DOS for each transfer made. It is, of course, completely independent of the physical sector size the disk uses.

To read a file, an application program passes to MS-DOS the size of a logical record, the first logical record to read, and the number of sequential logical records to read. Let's follow an example of how MS-DOS uses this information. Assume the application program is using 80-byte records and is set up to read a file 15 records at a time. Let's pick things up on its second read, that is, after it has already taken the first 15 records and is about to read the second 15. The request will be for an 80-byte record, the first record is number 15, and you want to read 15 records. Now pretend you're MS-DOS and analyze this request.

You immediately convert the request into byte-level operations. First multiply the logical record size by the record number, to get the byte position to start reading (80 x 15 = 1200). Then multiply the record size by the number of records, to get the number of bytes to transfer (also 1200).

Next, these numbers must be put in terms of physical disk sectors. This requires some divisions and subtractions involving the physical sector size and results in the breakdown of the transfer into three distinct pieces: (1) the position in the file of the first physical sector and the first byte within that sector to be transferred, (2) the number of whole sectors to transfer after the first (partial) sector, and (3) the number of bytes of the last (partial) sector to be transferred. The calculations and their results are shown in figure 6a. It is quite common for one or two of these pieces to be of length 0, in which case some of the following steps are not performed.

(6a)

$$\frac{\text{byte position 1200}}{\text{128 bytes/sector}} = 9, \text{ remainder } 48$$

Therefore, first byte to transfer is sector 9, byte 48

128 - 48 = 80 bytes to transfer in first sector

1200 - 80 = 1120 bytes left after first sector

$$\frac{\text{1120 bytes}}{\text{128 bytes/sector}} = 8, \text{ remainder } 96$$

Therefore, transfer 8 whole sectors, then 96 bytes of last sector

(6b)

$$\frac{\text{9 sectors}}{\text{4 sectors/allocation unit}} = 2, \text{ remainder } 1$$

Therefore, first sector to transfer is allocation unit 2, sector 1.

Figure 6: *Calculating physical, byte-level operations from logical definitions. The process outlined in figure 6a shows how the amount of data is calculated in physical terms. The actual position of data on the disk is computed in figure 6b.*

At this point, there is still no hint as to where the data will actually be found on the disk. You know that you want the tenth sector of the file (sector 9 because you start counting with 0), but you're not yet ready to check with the FAT to see where it is. The sector position in the file must be broken into two new numbers: the allocation unit position in the file and the sector within the allocation unit. For this example with single-density 8-inch disks (four sectors per allocation unit), this would be the third unit from the start of the file and the second sector within the unit (figure 6b).

What you need to do now is skip through the FAT to the third allocation unit in the file. If your file is the same one shown in figure 5, then from the directory entry you learn that the first unit is number 5. Looking at entry 5 of the FAT, you see the second unit is number 6. And finally, from entry 6 of the FAT, you find the third unit of the file, number 3. Table 2 reminds you that allocation unit number 3 is made up of physical sectors 34, 35, 36, and 37; therefore, physical sector 35 is what you are looking for.

Now the disk reads begin. Physical sector 35, only part of which is needed, is read into the single buffer kept in MS-DOS solely for this purpose. Then that part of the sector that is needed is moved as a block into the place requested by the application program.

Next, the whole sectors are read. MS-DOS looks ahead in the FAT to see if the allocation units to be transferred are consecutive. If so, they are combined into a single multiple-sector I/O system transfer request, which allows the I/O system to optimize the transfer. This is the primary reason why MS-DOS disks do not ordinarily use any form of sector interleaving: a well-written I/O system will be able to transfer consecutive disk sectors if told to do it in a single request. The overhead of making the request,

however, would often be too great to transfer consecutive sectors if it were done on a sector-by-sector basis.

Back to the example. MS-DOS will request that the I/O system read sectors 36 and 37 directly into the memory location called for by the application. Then, noting that allocation units 9 and 10 are consecutive, the corresponding sectors 58, 59, 60, and 61 from unit 9 and sectors 62 and 63 from unit 10 will be read by the I/O system in a single request. This completes the transfer of the eight whole sectors.

To finish the job, sector number 64 is read into the internal MS-DOS sector buffer. Its first 96 bytes are moved to the application program's area.

The Sector Buffer

This example shows the internal MS-DOS sector buffer being used in a very simple way. In reality, MS-DOS would normally perform the disk read in the example more efficiently than described here due to its optimized buffer handling. By keeping track of the contents of the buffer, disk accesses are minimized. The resulting speed improvement can be dramatic particularly when the requested transfer size is small (a fraction of a sector).

In the example, I assumed the application program was sequentially reading 15-record chunks (at 80 bytes per record) and had already completed the first such read. This would mean that sector 35 (the first one read in this example) would already be in the sector buffer because its first 48 bytes were needed for the previous read. MS-DOS would not reread this sector but instead would simply copy the remaining 80 bytes into the area designated by the application.

Likewise, when the application is ready to read the third chunk of the file, MS-DOS will find sector 64 already in the sector buffer. The last 32 bytes of the sector will be moved into place without a disk read.

For its own internal simplicity, MS-DOS has only one sector buffer. Between the 15-record reads, should the application request some other transfer that requires use of the buffer, then the buffer contents will be changed, and these optimizations are not possible. In this particular case, in which most of the disk transfer does not need the buffer, there will be very little difference in speed either way. Let's look at a different case where this optimization is practically essential.

Suppose the application wishes to write a file sequentially, one 16-byte record at a time. When the first record is written, MS-DOS simply copies the 16 bytes into the first part of the sector buffer. As each of the next seven records is written, it too is just copied into the appropriate position in the sector buffer. Again with a 128-byte sector of an 8-inch single-density format, the sector buffer would be full at this point. Upon attempting to write the ninth record, MS-DOS would find it needs to put the record in a different sector from the one currently in the buffer. The current buffer contents are marked "dirty," meaning they must be written to disk rather than discarded. MS-DOS does this and then moves the ninth 16-byte record into the buffer.

Note that MS-DOS did not write the sector buffer automatically after 128 bytes had been written to it. This is because the DOS has no notion of a sequential file: every disk transfer has an explicitly specified record position and record size. Thus, it does not think of the buffer as "full" -- for all it knows, the application program might back up and write the first 16 bytes over again. So the data is simply kept in the buffer until the file is closed or until the buffer is needed for something else.

Another optimization was taking place here that may have gone unnoticed. MS-DOS is always aware of the exact size of its files, and the assumption in the previous example was that this file was being newly written. Had it already existed, MS-DOS would have been forced to preread each sector into the sector buffer before copying any records into it. This is essential in case the program does random writes, intending to change only selected portions of the file. When the file is being extended (as in this case), the preread is not performed.

The possible outcome of this approach to buffer handling is that when the application program requests a write and is told it was successfully completed, the data may, in fact, not yet be written to the disk. The alternative approach is called *buffer write-through*, in which the data in the sector buffer would be written to disk each time the application requested a write. This would mean, in the example, eight rewrites of the same sector before moving on to the next, requiring a minimum of 1.2 seconds to write just 128 bytes! As the logical record size gets smaller, the time required to write becomes greater.

The presence of only one buffer does bring about the definite possibility of buffer "thrashing." Take the example of an application such as a compiler that will alternately read a small amount from one file and write a little bit to another. If both the reads and writes consist of a single 16-byte record, then the following sequence will be performed for each pair of records:

- read input file, get record from buffer
- read output file, put record into buffer
- write output file

For each record pair, three disk transfers are required. The result would be unbearably slow. The presence of only one sector buffer in MS-DOS is a design inadequacy that is difficult to defend (but it does help keep the DOS small). The practical solution is for applications that must access more than one file at a time to provide their own internal buffering. By requesting transfers that are at least half as big as the sector size, thrashing can be substantially reduced.

MS-DOS 2.0

Microsoft has now made available MS-DOS version 2.0 to all OEM (original equipment manufacturer) customers of previous versions. The 10 months put into version 2.0 by the MS-DOS team probably exceeds the total effort behind the previous 1.25 release, including the original development at Seattle Computer Products. While the changes have been substantial, the basic structure is still recognizable. I

have been discussing the DOS at such a low level that most of what I've talked about applies directly to version 2.0 as well. Here are the three main differences, along with my personal comment as original author of the DOS. I was not involved in the MS-DOS 2.0 project.

MS-DOS 2.0 allows multiple-sector buffers. The number is determined by a configuration file when the DOS is loaded. It can be easily adjusted to the user's needs: for example, to accommodate more buffers to prevent thrashing (this is the ideal solution to the buffer thrashing problem previously discussed) and fewer buffers to make more system memory available.

The new MS-DOS does not keep the file allocation tables in memory at all times. Instead, the tables share the use of the sector buffers along with partial-data transfers. This means that at any one time, all, part, or none of a FAT may be in memory. The buffer-handling algorithms will presumably keep often-used sectors in memory, and this applies to individual sectors of the FAT as well. This change in the DOS goes completely against my original design principles. Memory is getting cheaper all the time, so dedicating a few thousand bytes to the FATs should be completely painless. Now we're back to doing disk reads just to find out where the data is. In the case of a random access to a large fragmented file (for example, when accessing a database that fills half of a small Winchester disk), it is possible that several sectors of the FAT would need to be visited, in random order, to find the needed allocation unit.

While MS-DOS retains the original fixed-size main directory, it now can have files as subdirectories. This hierarchical (tree-structured) directory system may be extended to any depth. This approach is nearly essential for users to keep track of all the files that might be on a hard disk.

MS-DOS version 2.0 is, on the whole, a substantial upgrade of the previous releases. The three preceding paragraphs are intended only to point out the way the 2.0 file structure differs from the file structure I've discussed, not to give you a complete product description.

About the Author

Tim Paterson worked for Seattle Computer Products on the design of its 8086 computer system and the operating system now called MS-DOS. He then worked for Microsoft for about a year. Since returning to Seattle Computer Products as director of engineering, he has been primarily involved with new hardware development.