

Disk Filesystem Examples

John Leis

March 9, 2006

1 Overview

This document gives a very brief overview of the File Allocation Table (FAT) system, as used on Windows' floppy disks. It is not intended to be exhaustive, but rather as an instructive introduction as to how a filesystem might be designed and implemented. Other filesystems – such as NTFS, EXT2/EXT3 and FFS are in common use, on Windows, Linux, and FreeBSD systems, respectively.

When a “command shell” is required, click **start-run** and type **cmd**. Type **exit** to close the command shell. When converting from decimal to hexadecimal and binary, the Windows calculator may be used. Click **start-run** and type **calc**. It may be necessary to select **view-scientific** to enable hex/binary operations.

2 Getting the Sectors from a Disk

The “debug” program may be used to load sectors from a floppy disk. In a command shell, type **debug**, and press enter. At the “-” prompt, type “?” to see a list of commands. The most useful commands at present are the “l” (letter ell) command (to load a sector from disk into memory), and the “d” command (to dump the contents of the sector in hexadecimal).

Figure 1 shows how to load and dump a sector from a disk. The load command has arguments *address*, *drive*, *sector*, and *number of sectors*. In this case, we load into the default memory address of zero, from drive A: (logical drive zero). Exactly one sector is loaded — the boot sector (logical sector zero). Note the `segment:offset` format for the memory address, in the left-hand column. The exact location that debug loads the data into is unimportant (debug will choose some free memory for this). The dump command shows the first 256 bytes of the total of 512 bytes in the memory buffer.

3 Disk Layout

Figure 2 shows the layout of the disk after formatting. The boot sector is the very first sector of the disk, and contains:

Disk Geometry This contains information about the number of heads, sectors and cylinders on the disk, the sector size, and the “housekeeping” sectors of the disk.

Boot Code This is the code which loads the rest of the operating system. When the operating system starts up, the boot code is loaded and executed in order to load the remainder of the operating system.

Partition Table This may be present on hard disks to split the one physical drive into logical drives. This allows easier management of larger disks.

```

A:> debug
-1 0 0 0 1  load the boot sector
-d 0 ff
0AEE:0000 EB 3C 90 4D 53 44 4F 53-35 2E 30 00 02 01 01 00  .<.MSDOS5.0.....
0AEE:0010 02 E0 00 40 0B F0 09 00-12 00 02 00 00 00 00 00  ...@.....
0AEE:0020 00 00 00 00 00 00 29 56-07 56 4A 4E 4F 20 4E 41  .....)V.VJNO NA
0AEE:0030 4D 45 20 20 20 20 46 41-54 31 32 20 20 20 33 C9  ME FAT12 3.
0AEE:0040 8E D1 BC F0 7B 8E D9 B8-00 20 8E C0 FC BD 00 7C  ....{....|
0AEE:0050 38 4E 24 7D 24 8B C1 99-E8 3C 01 72 1C 83 EB 3A  8N$}$$.<.r...:
0AEE:0060 66 A1 1C 7C 26 66 3B 07-26 8A 57 FC 75 06 80 CA  f..|&f;.&.W.u...
0AEE:0070 02 88 56 02 80 C3 10 73-EB 33 C9 8A 46 10 98 F7  ..V....s.3..F...
0AEE:0080 66 16 03 46 1C 13 56 1E-03 46 0E 13 D1 8B 76 11  f..F..V..F....v.
0AEE:0090 60 89 46 FC 89 56 FE B8-20 00 F7 E6 8B 5E 0B 03  `F..V.. ..^..
0AEE:00A0 C3 48 F7 F3 01 46 FC 11-4E FE 61 BF 00 00 E8 E6  .H...F..N.a....
0AEE:00B0 00 72 39 26 38 2D 74 17-60 B1 0B BE A1 7D F3 A6  .r9&8-t.`....}..
0AEE:00C0 61 74 32 4E 74 09 83 C7-20 3B FB 72 E6 EB DC A0  at2Nt... ;r....
0AEE:00D0 FB 7D B4 7D 8B F0 AC 98-40 74 0C 48 74 13 B4 0E  .}.}....@t.Ht...
0AEE:00E0 BB 07 00 CD 10 EB EF A0-FD 7D EB E6 A0 FC 7D EB  .....}.....}.
0AEE:00F0 E1 CD 16 CD 19 26 8B 55-1A 52 B0 01 BB 00 00 E8  .....&.U.R.....

```

Figure 1: Using `debug` to display a disk sector. The “1” command loads the boot sector of drive A: into memory. The first 256 bytes are then shown in hexadecimal on the left, and as ASCII characters on the right.

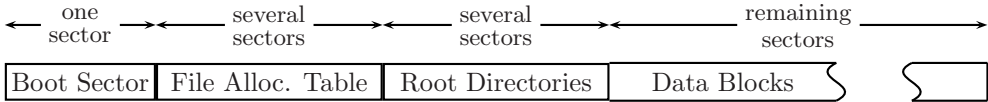


Figure 2: The disk layout. The boot sector contains the disk geometry and boot code. This is followed by the file indexing information, while the file data blocks constitute the remainder of the disk.

Following the boot sector is the **file allocation table** or FAT. This maps the allocation of the various disk sectors to files — in effect, it is a “mapping table” for the file’s contents onto the physical sectors on the disk.

The file allocation will vary depending on the operating system (actually, the *filesystem* in use), and may not necessarily be contiguous as shown in Figure 2. However, the FAT file system has a contiguous section dedicated to the file allocation table. In fact, it may not be called a “file allocation table” at all in different filesystem implementations. DOS/Windows often stores more than one identical copy of the file allocation table in this section.

Next are the **root directory entries**. These store the directory entries for each file in the root directory. The most important data in each directory entry is the name of the file, and a way to access the disk sectors used for each file. Of course, other information such as the file creation time may be stored as well. Subdirectories are stored in a different manner, alongside normal files. This will be explained later.

Finally, the **data blocks** store the data belonging to all the files, and the subdirectories below the root level. The data blocks are fixed in size, and are always a small integer number of sectors. This is termed a **cluster** in the FAT filesystem. On floppy disks, one cluster is exactly one sector (512 bytes). Typically, a cluster is 1, 2, 4, ... 32 sectors, depending on the size of the disk.

3.1 Data Structures

The information in user files may be any combination of bytes, as determined by the application. At the disk-sector level, it is necessary to impose structure on the raw data stored on a disk’s sectors when saving configuration parameters. For example, the number of bytes stored on each disk sector is stored in the boot sector, and must be

interpreted correctly (as a 16-bit number).

Structured data may be single bytes, multi-byte constants, or data structures of related information. For multi-byte constants, little-endian ordering (low-order byte at the lowest memory address) is used. As will be seen later, data structures may contain single or multi-byte constants, or bitfields encoded within one or more bytes.

The specific data represented may be stored in a group of bits which does not correspond to an integral number of bytes. For example, the date and time require sub-fields of year, month, day, hour, minute and second. The day of the month requires 31 possible values, hence a 5-bit number is sufficient.

Figure 3 lists the terminology used in this document. Integer constants are assumed to be base-10. The C-language terminology of preceding hexadecimal (base-16) constants with “0x” is used. Thus, 45 is the base 10 number 45, whereas 0x45 is the hexadecimal number 45 (or $4 \times 16 + 5 = 69$ base 10).

Data type	Description
character	8-bit bytes using ASCII character encoding.
byte	8-bit unsigned value
word	16-bit integer
dword	32-bit integer (doubleword)
data structure	Related data, occupying several bytes.

Figure 3: The data type names used to describe the various basic data types and compound data types. Integers are normally unsigned.

3.2 The Boot Sector

The physical layout of the boot sector is shown in Figure 4. In order for the operating system to boot, the boot code is executed by loading the boot sector into memory from disk, and jumping to the first memory location. This instruction is, in turn, a jump to the boot code stored later in the first sector (at offset 0x1E).

The geometry of the disk is stored as various constants between the first jump instruction and the boot code itself. As well as the sector size, the number of sectors, the number of tracks (or cylinders), and the number of disk heads, the number of reserved sectors for use in locating information on the disk itself is defined.

The partition table information is shown in Figure 5. It is not present on floppy disks because of their small capacity. It allows one physical drive (for example, c:) to be split into (say) c: and d: drives for easier management or backup purposes. If the partition is inactive, its status is set to 0. If it is bootable, its status is 0x80. The partition’s start & end of sector and cylinder are encoded as per Figure 6.

To illustrate the interpretation of the boot sector values, Figure 7 shows the result of formatting a disk using the FAT file system. This shows the quantity 0x0200 (512 decimal) as the number of bytes per sector, and 0x0B40 (2880) as the number of sectors. Thus the unformatted capacity of the disk is

$$512 \frac{\text{bytes}}{\text{sector}} \times 2880 \text{ sectors} = 1474560 \text{ bytes}$$

This should be compared to Figure 8, which shows that the number of bytes per sector is 512, but that the number of sectors (“allocation units”) available is less than that expected. This is because of the space used by the boot, FAT, and root directory sections.

4 Storing Files

There are two key components to storing a file (or directory) in the file allocation table (FAT) file system:

Data Type	bytes	Description
op-code	3	jump to boot code instruction
character	8	Manufacturer name & Version
word	2	Bytes per sector
byte	1	Sectors per cluster
word	2	Number of reserved sectors
byte	1	Number of FAT tables
word	2	Number of root directory entries
word	2	Number of sectors
byte	1	Media descriptor
word	2	Number of FAT sectors
word	2	Sectors per track
word	2	Number of heads
word	2	Number of hidden sectors
op-codes	416	Boot code
data structure	16	Partition information (hard disk)
bytes	50	Unused/boot
Total:	512	

Figure 4: The layout of the boot sector. As well as the physical geometry of the disk (sectors, tracks/cylinders and sides/heads), the allocation of disk sectors to indexing information is present. The boot code is normally only present for bootable disks — if the disk is not bootable (i.e. cannot load an operating system), then the bytes reserved for the boot code contain meaningless information.

Data type	bytes	Description
byte	1	Partition status
byte	1	Start Head
bitwise data structure	2	Start Sector & Cylinder
byte	1	Type
byte	1	End Head
bitwise data structure	2	End Sector & Cylinder
word	2	Start logical sector
word	2	Number of sectors
Total:	12	

Figure 5: The partition table layout. The partition table is only present on large disks, to allow the subdivision of one physical disk into smaller logical disks.

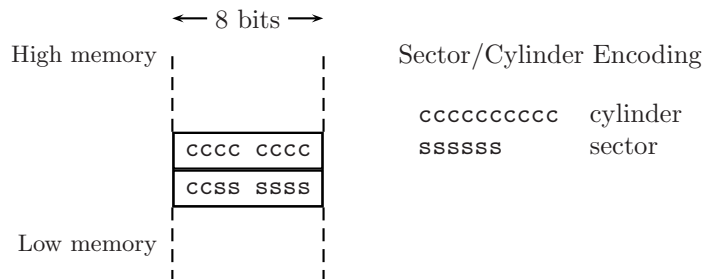


Figure 6: Decoding sector/cylinder values in the partition table. Six bits are used for the sector, and ten bits are used for the cylinder (track). Both the start and end of the partition are encoded in this way.

```

A:> debug
-1 0 0 0 1  load the boot sector
-d 0 1d
OAE:0000 EB 3C 90 4D 53 44 4F 53 - 35 2E 30 00 02 01 01 00
OAE:0010 02 E0 00 40 0B F0 09 00 - 12 00 02 00 00 00

```

number of sectors *bytes per sector*

Figure 7: Decoding the boot sector. The raw byte fields are shown using **debug**, and correspond exactly with the byte alignment and data size shown in Figure 4.

```

C:\>format a:
Insert new disk for drive A:
and press ENTER when ready...
The type of the file system is FAT.
Verifying 1.44M
Initializing the File Allocation Table (FAT)...
Volume label (11 characters, ENTER for none)?
Format complete.
    1,457,664 bytes total disk space.
    1,457,664 bytes available on disk.
        512 bytes in each allocation unit.
    2,847 allocation units available on disk.
    12 bits in each FAT entry.

```

Figure 8: Formatting a disk. The filesystem parameters are determined by the filesystem allocation method, which is influenced by the total size of the disk media.

The directory entry stores the file name and index of the starting sector, together with various other information about the file.

The file allocation table stores the indexing information for subsequent sectors used by the file. This is needed because any given file may occupy non-contiguous sectors on the disk.

The FAT indexes all usable sectors on the disk — that is, all sectors not used for boot information, the file table itself, or root directory information. If a sector is unused and is available for allocation, the index value indicates this fact by setting the FAT entry to 0. If a sector is the last sector belonging to a file, the index is set to a value between FF8 and FFF. The value FF7 is present if the corresponding sector is bad and should not be used, as occasionally errors in the magnetic media render one or more sectors unable to be used.

4.1 File Entries

Files use a data structure as illustrated in Figure 9. Each directory entry contains the name of the file or directory, its type and size, and a pointer to the starting file table entry. The file table entries comprise what is called a “linked list”. Starting with the first entry, each subsequent entry contains the index of the entry which follows it. The entry number corresponds to the sector number. In the case of floppy disks, the indexes stored at each entry are 12-bit numbers (for larger disks they are 16 or 32 bits).

As a numerical example, consider Figure 9, which shows a hypothetical file allocation. The directory entry has a pointer to the first file table entry, and in this case has a value of 10. This means that that physical sector 10 contains the first 512 bytes of the file¹.

¹For the sake of simplicity, we are referring to sectors here, but strictly speaking the indexes are used to index clusters (contiguous groups of

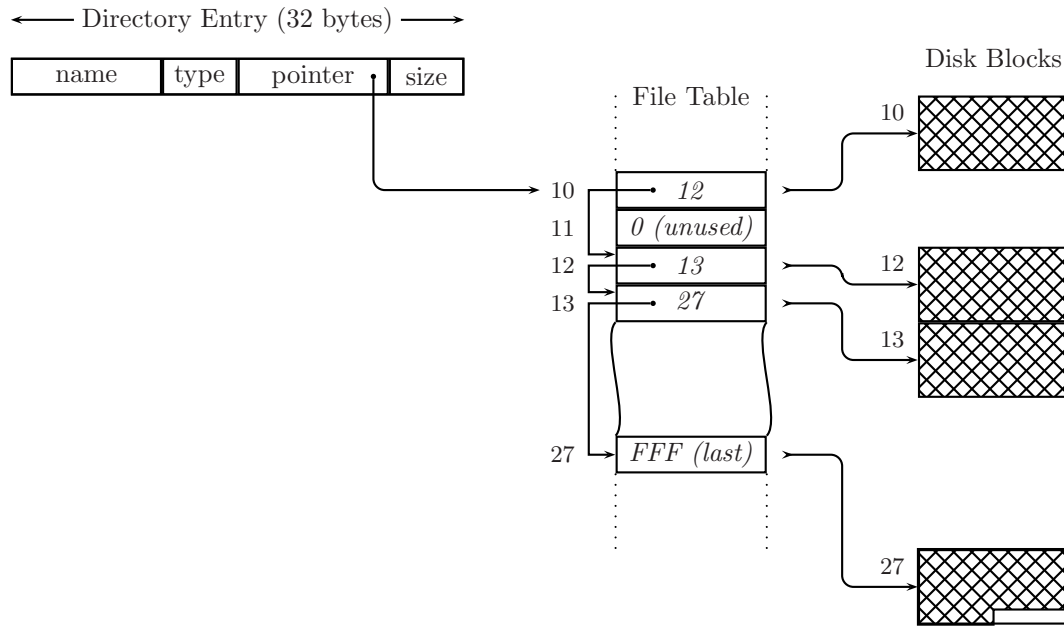


Figure 9: Disk block indexing. The directory entry is stored in either the root directory sectors of the disk, or in a disk block if that block corresponds to a subdirectory (and not a regular file).

Data type	bytes	Description
character	8	Name
character	3	Extension
byte	1	Attribute
byte	10	reserved
bitwise data structure	2	Time
bitwise data structure	2	Date
word	2	Starting sector
dword	4	Size
Total:	32	bytes

Figure 10: The directory entry layout. These are contained in either the root directory sectors of the disk, or on other disk blocks if the disk block is marked as containing a directory.

The number stored at index 10 in the table is 12. The next 512 bytes of the file are stored in sector 12. Index 12 in the allocation table contains the value 13, meaning sector 13 contains the next 512 bytes. Index 13 contains the value 27. The subsequent bytes are contained in sector 27. However, this sector is not completely full, even though a full sector is allocated. Index 27 in the table contains all-ones (the 12-bit number FFF), which indicates that this is the last entry in the chain, and therefore the last sector of the file. The last sector may contain between one and 512 bytes.

As stated above, the index of the FAT entry corresponds to the cluster number (which is, in turn, the sector number for small disks such as floppies). This is not entirely correct, as the FAT indexes start at two, and the first available data sectors start after the root directory entries. However, this is simply a matter of an appropriate offset being added.

The byte allocation in each directory entry is shown in Figure 10. Note that in older systems, each file name could only be eight characters, with a three-character extension, because of the amount of space reserved to store the name.

In this particular disk, the boot sector shown in Figure 11 indicates that there are two FAT tables, and 9 sectors for each FAT table, for a total of $2 \times 9 = 18$ sectors used. Of course, one additional sector is used at the start for the boot sector. If we give a volume label to the disk and dump sector 19 (= 0x13), we can see in Figure 12 that the very first sectors). In a floppy disk, one sector is identical to one cluster.

```

A:> debug
-1 0 0 0 1  load the boot sector
-d 0
0AEE:0000 EB 3C 90 4D 53 44 4F 53-35 2E 30 00 02 01 01 00 .<.MSDOS5.0.....
0AEE:0010 02 E0 00 40 0B F0 09 00-12 00 02 00 00 00 00 00 ...@.....
0AEE:0020 00 00 00 00 00 00 29 56-07 56 4A 4E 4F 20 4E 41 .....)V.VJNO NA
0AEE:0030 4D 45 20 20 20 20 46 41-54 31 32 20 20 20 33 C9 ME FAT12 3.
0AEE:0040 8E D1 BC F0 7B 8E D9 B8-00 20 8E C0 FC BD 00 7C ....{....|
0AEE:0050 38 4E 24 7D 24 8B C1 99-E8 3C 01 72 1C 83 EB 3A 8N$}$....<.r...:
0AEE:0060 66 A1 1C 7C 26 66 3B 07-26 8A 57 FC 75 06 80 CA f..|&f;.&.W.u...
0AEE:0070 02 88 56 02 80 C3 10 73-EB 33 C9 8A 46 10 98 F7 ..V....s.3..F...

```

Figure 11: A hexadecimal dump of the first 128 bytes of the 512 bytes contained in the boot sector of a disk. Most of the space is reserved for the boot code, but the disk and filesystem parameters are of crucial importance.

```

A:> label mydisk
A:> debug
-1 0 0 13 1  load the root directory sector
-d 0
0AEE:0000 4D 59 44 49 53 4B 20 20-20 20 20 08 00 00 00 00 MYDISK .....
0AEE:0010 00 00 00 00 00 00 56 78-FD 2E 00 00 00 00 00 00 .....Vx.....
0AEE:0020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0030 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....

```

Figure 12: Storing the label or volume name of a disk. It is stored as a regular directory entry, with bit 3 of the attribute byte set to indicate that it is a volume name and not a regular file.

directory entry contains the volume label.

Because there are 224 root directory entries (0x00E0), each occupying 32 bytes, the number of sectors occupied by the root directory entries is

$$\frac{224 \text{ entries} \times 32 \frac{\text{bytes}}{\text{entry}}}{512 \frac{\text{bytes}}{\text{sector}}} = 14 \text{ sectors}$$

The attribute byte (Figure 13) is a set of bit flags, which tells (among other things) whether a directory entry is actually a regular file, a directory, or the name of the disk volume. The directory entry in this case has an attribute byte of 0x08. This indicates bit 3 is set, showing that it is indeed a volume label and not a regular file. The date and time encoding are shown in Figure 14. These are packed into appropriately-sized bitfields.

To see how the directory entry for a regular file is stored, we create and copy a small file of 35 bytes. The first directory sector now contains the volume label, followed by the directory entry for this file, as shown in Figure 15.

4.2 Indexing Files

Once we have the starting index, we must find all subsequent sectors used by a particular file. On a floppy disk, the indexes are stored as 12-bit numbers, encoded as in Figure 16.

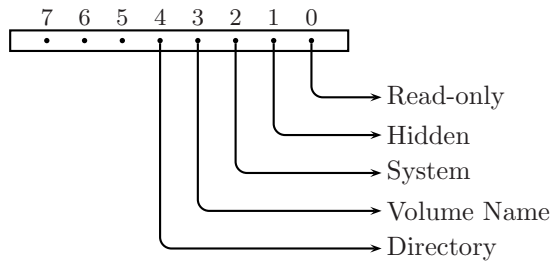


Figure 13: Decoding the attribute byte. Individual bits act as “flags” for various parameters, including the fact that the “file” may in fact be the volume name, or a directory entry. If the latter, the “data” portion of the “file” actually contains directory entries, not file data.

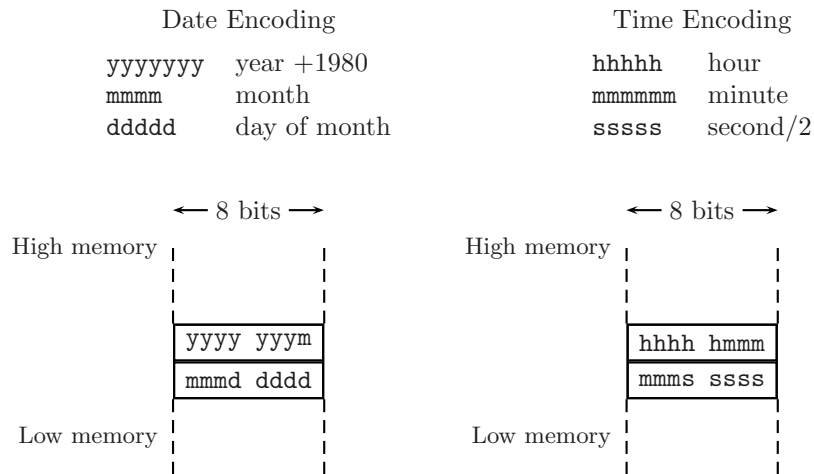


Figure 14: Decoding date and time values. The quantities are packed into bytes as shown. Note that the year is relative to 1980. This is a common practice — Unix systems generally store the year relative to 1970. This fact, in conjunction with the number of bits used to store the year, imposes a maximum on the year which may be represented, which may be problematic in some circumstances.

```

A:> debug
-1 0 0 13 1  load the root directory sector
-d 0
0AEE:0000 4D 59 44 49 53 4B 20 20-20 20 20 08 00 00 00 00 MYDISK .....
0AEE:0010 00 00 00 00 00 00 72 78-FD 2E 00 00 00 00 00 .....rx.....
0AEE:0020 46 49 4C 45 31 20 20 20-54 58 54 20 18 AC A8 78 FILE1 TXT ...x
0AEE:0030 FD 2E FD 2E 00 00 A5 78-FD 2E 02 00 23 00 00 .....x...#...
0AEE:0040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 .....
0AEE:0050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 .....
0AEE:0060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 .....
0AEE:0070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 .....

```

Figure 15: The first root directory sector on the disk. The first entry is the volume name (disk label) MYDISK, followed in this case by the file named FILE1.TXT. The associated directory entry contains the file date, time, and starting FAT entry.

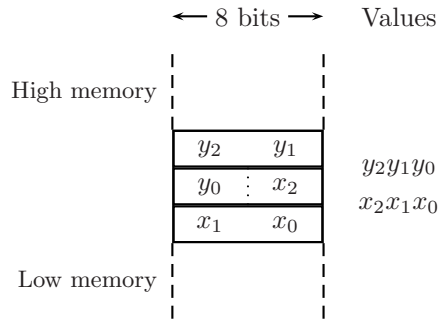


Figure 16: Decoding 12-bit FAT entries. The 12-bit layout is consistent with little-endian byte ordering. Larger disks use 16 or 32 bits for the block indexes.

```

A:> debug
-1 0 0 1 1  load the file allocation sector
-d 0
0AEE:0000  F0 FF FF 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0AEE:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0AEE:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0AEE:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0AEE:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0AEE:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0AEE:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0AEE:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....

```

Figure 17: The initial state of the first FAT sector. The first 12 bits indicates that this disk media is a 1.44M disk. No files are stored as yet, hence the subsequent FAT entries are all zero.

The first three bytes in the FAT table are not used for normal entries. The first entry is in fact the disk type, and the second is a padding value to use up the remaining 12 bits so that the subsequent entry starts on a byte boundary. Figure 17 shows the FAT table of an empty filesystem. The value FF0 is present here, to indicate a 1.44M disk. It would be FF8 for a hard disk.

Next, we copy a 35-byte file. This occupies one sector (by definition) of the disk. To see which sector is used, we examine the directory entry again, as shown in Figure 18.

The FAT table (Figure 19) shows that the entry for sector 2 is FFF, and hence the sector is occupied. Furthermore, it is the last sector for that file. Note that the starting FAT index is 2, even though it refers to the first logical sector in the data-block portion of the disk. The dump of the first (and only) data sector belonging to this file is shown in

```

-1 0 0 13 1  load the root directory sector
-d 0
0AEE:0000  4D 59 44 49 53 4B 20 20-20 20 20 08 00 00 00 00 MYDISK .....
0AEE:0010  00 00 00 00 00 00 00 72 78-FD 2E 00 00 00 00 00 .....rx.....
0AEE:0020  46 49 4C 45 31 20 20 20-54 58 54 20 18 AC A8 78 FILE1  TXT ...x
0AEE:0030  FD 2E FD 2E 00 00 A5 78-FD 2E 02 00 23 00 00 00 .....x...#...
0AEE:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....

```

Figure 18: The first root directory sector on the disk. The first entry is the volume name (disk label) MYDISK, followed in this case by the file named FILE1.TXT.

```

-1 0 0 1 1  load the file allocation sector
-d 0
0AEE:0000  F0 FF FF FF 0F 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....

```

Figure 19: The state of the FAT sector after copying one file, occupying only one sector. The remaining sectors are marked as “free” by using an index value of zero.

```

-1 0 0 21 1  load the first file data sector
-d 0
0AEE:0000  54 68 69 73 20 69 73 20-66 69 6C 65 20 6F 6E 65  This is file one
0AEE:0010  0D 0A 74 68 65 20 6C 61-73 74 20 6C 69 6E 65 0D  ..the last line.
0AEE:0020  0A 0D 0A 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
0AEE:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....

```

Figure 20: The data portion of FILE1.TXT. Even though the file is small, an entire sector is allocated.

Figure 20. A second file of 541 bytes is then copied. The FAT table and data sectors are shown in Figure 21. Now, index 2 holds the value FFF to show that it is the last sector. Index 3 holds the value 4 to indicate that the next sector in that chain is 4, and index 4 holds FFF, to indicate the end of the chain.

5 Obtaining Low-level Disk Information under Unix/Linux/FreeBSD

The preceding sections described the use of **debug** under the Windows command shell to obtain the disk sector information directly. This can of course be done under Unix. The following briefly describes the necessary commands. Of course, the low-level data structures of a FAT formatted disk are the same as described earlier.

Under Unix, devices may be accessed directly via the **/dev** directory. Once a filesystem is mounted, it becomes accessible via the appropriate *mount point*. For example, disk partition 3 on device hda (the normal C: drive) may be mounted under Linux using

```
mount -t vfat /dev/hda3 /mnt/share
```

FreeBSD uses a different naming convention for hard disk access, using both “partitions” and what are termed “slices”². So for example, to access the second IDE drive (ad1), slice 2, partition ‘a’, **mount** would be invoked using

```
mount -t msdosfs /dev/ad1s2a /mnt/share
```

²Actually a BSD “slice” corresponds to a conventional partition, and BSD partitions are created within each slice.

Floppies are named similarly under both Linux and FreeBSD, being designated `/dev/fd0` for the A: floppy. To create a FAT filesystem under Linux on a floppy, we can use the “makefilesystem” command:

```
mkfs -t vfat /dev/fd0
```

To access files we can *mount* the filesystem as follows:

```
mount -t vfat /dev/fd0 /mnt/floppy
```

In order to do a direct disk dump, as was done with **debug** in Windows, it is necessary to use the **dd** (disk dump) command. To dump the first physical 512-byte sector on floppy A, use

```
dd bs=512 count=1 if=/dev/fd0 skip=0
```

This gives a character dump. For the purposes of interpretation, it may be necessary to convert the output into ASCII characters or hex bytes. This is done using the **od** command, with appropriate command-line flags. Figures 22 and 23 illustrate this³.

Normally root access is required for low-level access to disks under Unix. This is for good reason, since inadvertently changing an index block or the boot sector can lead to catastrophic results. **Never** use **of** in place of **if** in the **dd** command above, since this will lead to a disk *write* instead of a disk *read*, almost certainly resulting in a corrupted and unusable disk.

The above examples are meant to outline the essential commands for this task. Use **man** to find out more about the command formats and option flags.

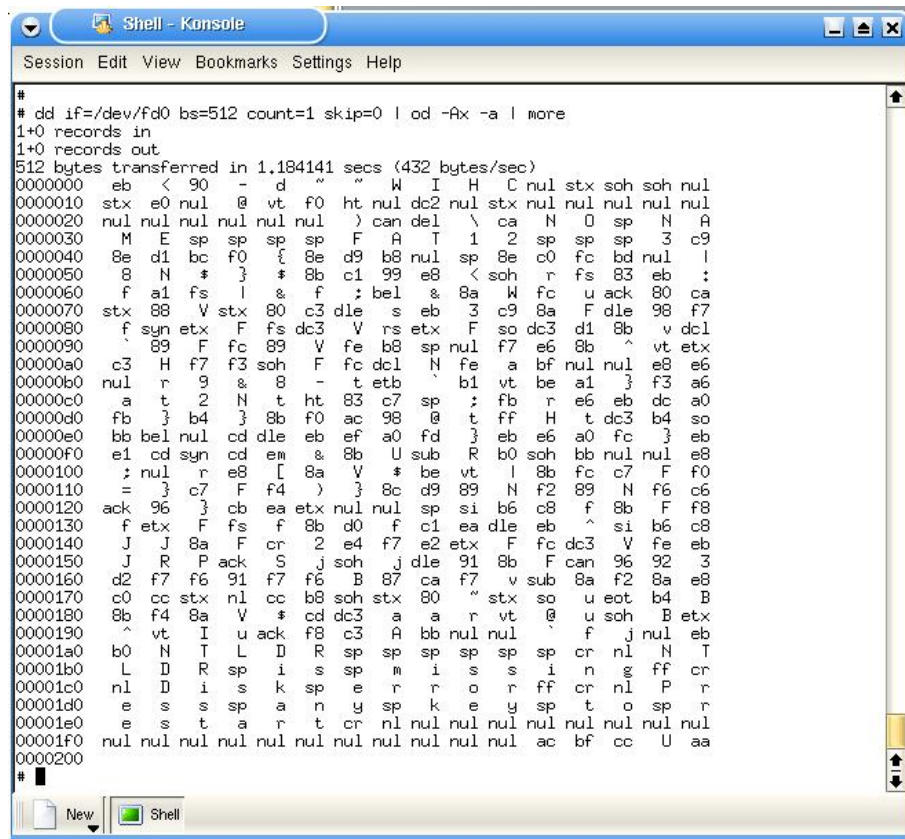


Figure 22: Using diskdump tools in FreeBSD/Linux.

³These were produced using the **ksnapshot** tool under the KDE desktop manager.

```

# dd if=/dev/fd0 bs=512 count=1 skip=0 | od -Ax -x | more
1+0 records in
1+0 records out
512 bytes transferred in 1.185517 secs (432 bytes/sec)
0000000  3ceb  2d90  7e64  577e  4849  0043  0102  0001
0000010  e002  4000  f00b  0009  0012  0002  0000  0000
0000020  0000  0000  0000  1829  5c7f  4eca  204f  414e
0000030  454d  2020  2020  4146  3154  2032  2020  c933
0000040  d18e  f0bc  8e7b  b8d9  2000  c08e  bdfc  7c00
0000050  4e38  7d24  8b24  99c1  3ce8  7201  831c  3aeb
0000060  a166  7c1c  6626  073b  8a26  fc57  0675  ca80
0000070  8802  0256  c380  7310  33eb  8ac9  1046  f798
0000080  1666  4603  131c  1e56  4603  130e  8bd1  1176
0000090  8960  fc46  5689  b8fe  0020  e6f7  5e8b  030b
00000a0  48c3  f3f7  4601  11fc  fe4e  bf61  0000  e6e8
00000b0  7200  2639  2d38  1774  b160  be0b  7da1  a6f3
00000c0  7461  4e32  0974  c783  3b20  72fb  ebe6  a0dc
00000d0  7dfb  7db4  f08b  98ac  7440  480c  1374  0eb4
00000e0  07bb  cd00  eb10  a0ef  7dfd  e6eb  fca0  eb7d
00000f0  cde1  cd16  2619  558b  521a  01b0  00bb  e800
0000100  003b  e872  8a5b  2456  0bbe  8b7c  c7fc  f046
0000110  7d3d  46c7  29f4  8c7d  89d9  f24e  4e89  c6f6
0000120  9606  cb7d  03ea  0000  0f20  c8b6  8b66  f846
0000130  0366  1c46  8b66  66d0  eac1  eb10  0f5e  c8b6
0000140  4a4a  468a  320d  f7e4  03e2  fc46  5613  ebfe
0000150  524a  0650  6a53  6a01  9110  468b  9618  3392
0000160  f7d2  91f6  f6f7  8742  f7ca  1a76  f28a  e88a
0000170  ccc0  0a02  b8cc  0201  7e80  0e02  0475  42b4
0000180  f48b  568a  cd24  6113  7261  400b  0175  0342
0000190  0b5e  7549  f806  41c3  00bb  6000  6a66  eb00
00001a0  4eb0  4c54  5244  2020  2020  2020  0a0d  544e
00001b0  444c  2052  7369  6d20  7369  6973  676e  0dff
00001c0  440a  7369  206b  7265  6f72  ff72  0a0d  7250
00001d0  7365  2073  6e61  2079  656b  2079  6f74  7220
00001e0  7365  6174  7472  0a0d  0000  0000  0000  0000
00001f0  0000  0000  0000  0000  0000  ac00  ccbf  aa55
0000200
#

```

Figure 23: Using diskdump tools in FreeBSD/Linux.

References & Further Reading

- [1] John Angermeyer *et al*, *MS-DOS Developer's Guide*, Waite Group/Howard W. Sams & Co., 1989.
- [2] Jonathan Fox, *FAT System Guide*, web reference, 2003,
<http://home.freeuk.net/foxy2k/disk/disk1.htm>
 current March 2006.
- [3] Neil Brown, *Who wants another Filesystem?*, Fourth Annual Linux Conference, 2003,
<http://www.cse.unsw.edu.au/~neilb/conf/lca2003/>
 current March 2006.
- [4] *NTFS - New Technology File System*, ntf.com, 2003,
<http://www.ntfs.com/>
 current March 2006.
- [5] Scott Mueller, *Upgrading and Repairing PC's*, Que, Indianapolis, IN.
- [6] Michael Tischer, *PC System Programming*, Abacus, Grand Rapids, MI.