Delphi Internals: How not to Write an Operating System (3)

Disk formatting basics

by Dave Jewell

The naive might be forgiven for supposing that MS-DOS contains a simple 'Format Disk' call which takes a drive number, an indication of the required capacity of the disk and a few flags to specify whether you want to do a quick format and/or create a bootable system disk, provide a volume label and so forth. Those who believe in the Tooth Fairy might even anticipate that this magic call would work under Windows too...

Well, no such luck I'm afraid. It's frankly absurd that even though DOS is now up to 7.0 (the version which ships with Windows 95) the ability to format disks is still not built into the operating system. In the days before Windows, this wasn't unreasonable. The amount of code required to do the job is quite sizeable and it wouldn't have been logical to have that code hanging around in memory all the time doing nothing but eating into DOS's precious 640Kb of RAM. However, once Windows started taking off, Microsoft should have built the disk formatting code into a separate code segment of the KERNEL library (the logical place for it) so that a Windows application wishing to format a disk would be able to do so with one simple call.

Under Windows 3.1, the File Manager still has to do the job itself: around 7Kb of code is devoted purely to the job of formatting disks. Even under Windows 95, Microsoft still haven't provided a simple, all-in-one approach to the job of disk formatting. OK, they've made a nod in the direction of disk formatting by providing a new `DeviceIOControl` call which formats a specified number of tracks, but this is only a relatively small part of the entire formatting process. And, of course, `DeviceIOControl` is only available to 32-bit applications...

Just to add insult to injury, it turns out that there is a new routine called `SHFormatDrive`. This routine is implemented in the new 32-bit SHELL32.DLL library and it's used by the Windows 95 Explorer. However, it's completely undocumented and by the time you've reverse engineered it and figured out what parameters it takes, you may as well have written your own code. Thanks Microsoft...

As with the preceding parts of this series on low-level disk I/O, the emphasis here is on formatting disks from a 16-bit application, but retaining compatibility for 32-bit apps. Without documentation on the `SHFormatDrive` routine, things are a little bit easier for 32-bit programs, but only a little bit...

Five Floppy Format Flavours...

As I'm sure you'll appreciate, a floppy disk is made up of a number of concentric tracks, each of which has a number of sectors on it. Before a disk can be used, the track and sector information has to be laid down on the disk. Once that's done, we have to write a valid BPB into the first sector of the disk. The BPB specifies various characteristics of a disk – we'll discuss it in more detail presently.

The formatting program also needs to set up one or two FATs (File Allocation Tables) on the disk. These are what DOS uses to keep track of how each file is allocated. The FAT allows DOS to determine which sectors are occupied by a file, and in what order. As a file is read from disk, DOS steps through the appropriate entries of the FAT to determine the next sector it needs to read.

On top of all this, we also need to set up an empty directory on the disk. The directory contains an entry for each file on the disk and has other, special, entries which relate to sub-directories and the disk's volume label, if present – we touched on this a couple of months ago.

Life would be relatively straightforward if floppy disks only ever came in one flavour. However, we need to support at least five different types of disk capacity: 360Kb and 1.2Mb (5.25 inch) along with 720Kb, 1.44Mb and 2.88Mb (3.5 inch).

You'll remember I mentioned last month that the Editor won't tolerate slackers? Well, He Who Must Be Obeyed said that it would be nice if the software included a Quick Format option. This means that rather than physically formatting each track, the software simply writes a "factory fresh" BPB, FAT information and empty directory to the disk. The problem with this approach is that when the user selects Quick Format, we need to be able to auto-detect whatever format is already on the disk, involving more complexity. Is it all beginning to sound a bit more complicated than you first thought? Well you're right – it is!

If you look at Table 1, it lists the disk characteristics for each of the five floppy disk sizes mentioned above. In actual fact, there are a couple of older, single-sided formats (160Kb and 180Kb) and a 320Kb format which apply to 5.25 inch disks only. I considered whether to support these, but in the end I decided not to bother. Let's be realistic: the year is 1996 and anyone who is still using one of those old disk drives really needs to get a life!

Disk Size	720Kb	1.44Mb	2.88Mb	360Kb	1.2Mb
Heads	2	2	2	2	2
Tracks	80	80	80	40	80
Sectors/Track	9	18	36	9	15
Total Sectors	1440	2880	5760	720	2400
Free Sectors	1426	2847	5726	708	2371
Sectors/Cluster	2	1	2	2	1
Total Clusters	713	2847	2863	354	2371
Sectors/FAT	3	9	9	2	7
FAT Copies	2	2	2	2	2
Root Dir Sectors	7	14	15	7	14
Reserved Sectors	1	1	1	1	1
Hidden Sectors	0	0	0	0	0
Bytes/Sector	512	512	512	512	512
Bytes/Cluster	1024	512	1024	1024	512
Root Dir Entries	112	224	240	112	224
Media Descriptor	$F9	$F0	$F0	$FD	$F9

➤ *Table 1: Floppy disk characteristics*

```
type
  DiskType = record
    pc  : Byte;        { sectors per cluster        }
    rde : Integer;     { number of root-dir entries }
    sec : Integer;     { total number of sectors    }
    med : Byte;        { media descriptor           }
    spf : Integer;     { number of sectors per FAT  }
    spt : Integer;     { sectors per track          }
  end;
const
  { This array maps a logical drive type to a list of      }
  { parameters for that drive. Assumptions:                }
  { Bytes per sector = 512  Reserved sectors = 1           }
  { Number of FATS = 2      Heads = 2                      }
  { Hidden sectors = 0      Tracks = 80 except 40 for 1st  }
  DiskTypes: array [0..4] of DiskType = (
    (spc:2; rde:112; sec: 720; med:$FD; spf:2; spt: 9),   { 360 K }
    (spc:1; rde:224; sec:2400; med:$F9; spf:7; spt:15),   { 1.2 M }
    (spc:2; rde:112; sec:1440; med:$F9; spf:3; spt: 9),   { 720 K }
    (spc:1; rde:224; sec:2880; med:$F0; spf:9; spt:18),   { 1.4 M }
    (spc:2; rde:240; sec:5760; med:$F0; spf:9; spt:36));  { 360 K }
```

➤ *Listing 1*

Still referring to Table 1, you'll see that the five supported formats all have a few things in common. The physical size of a sector is always 512 bytes, disks are always double-sided, the reserved sector count is always one, there are always two FATs and the hidden sector count is always zero. In addition, the track count is 80 for everything except 360Kb disks. With all this in mind, I produced the lookup table shown in Listing 1. It maps a logical drive type (in the range 0..4) onto a set of disk parameters which define that drive's geometry.

Absolutely Fabulous Disk I/O

Of course, in order to quick format an existing volume, we need to know what format it is in the first place. To do this, the simplest approach is to simply read the first sector of the disk and examine the BPB to be found there. Shortly, we'll take a look at the structure of a BPB, but for now, let's concentrate on how to read that first sector. The various file I/O routines provided as part of the Delphi run-time library are just that: file oriented. They can't be used to read absolute sectors from the disk. To read an absolute sector number, we have to use a special interrupt number, `INT $25`. This interrupt allows us to treat a disk as a contiguous array of sectors. For example, a 720Kb floppy uses nine sectors per track so sector zero corresponds to the very first sector on the disk. Sector eight corresponds to the last sector on track zero and sector nine is the number of the first sector on track one. `INT $25` will allow us to randomly read any sector we want but (for the purposes of disk formatting) we're generally only interested in the first few sectors on the disk.

In a similar way, another routine, `INT $26`, can be used to perform absolute sector writes to a floppy disk. The code for two new routines, `AbsRead` and `AbsWrite`, is given in Listing 2. These two routines rely on the "old" form of `INT $25` and `INT $26`. That is to say, they won't work with disk partitions greater than 32Mb in size. For such large disks, you'd need to use a newer variant of these routines. Again, this isn't a problem for our purposes since we're developing code which is specific to floppy disks.

`AbsRead` and `AbsWrite` are quite straightforward to use, but if you're at all worried about accidentally zapping your hard disk (!) while experimenting with these routines, then I'd advise you to add code which checks that the drive number always corresponds to a floppy disk, ie in the range 1..2, on entry to these routines. Similar arguments apply to some of the other routines that we shall look at later. Both routines return an error code if they fail, or zero on success.

BPBs And All That...

Now that we can read absolute disk sectors, let's look in more detail at the structure of a BPB. As mentioned already, BPBs are located on the very first sector of a disk. They have to be located there since it's the BPB which specifies how to read the rest of the disk. Since BPBs form an integral part of the

boot sector (the first disk sector) I haven't shown a separate data structure for BPBs. Instead, look at the BootSector structure in Listing 3. This isn't the cleanest data structure in the world. It contains a lot of 'ifs' and 'buts' and frankly it's a bit of a mess. This is a reflection of the fact that it slowly evolved from the very earliest days of DOS. For example, the real BPB actually starts with the third field (bsBytesPerSec) and ends half way through the bsHiddenSectors field! The later fields such as the volume ID (the DOS serial number), volume label and file system identifier aren't present with early disks but form part of what's called the extended boot record.

In order to distinguish an extended boot record from a plain vanilla one, you have to look at the bsBootSignature field. If this contains the value $28 or $29, then you're looking at an extended boot record and the three aforementioned fields are valid. Otherwise, you're not. You'll also notice that the bsHugeSectors field (which specifies the total number of sectors on disk) is only valid if the bsSectors field is zero. This, of course, is to support hard disks with a larger sector count than can be contained in a 16-bit integer.

Keep in mind that this data structure is equally applicable to floppy disks and hard disks. The only real difference is that on a floppy disk, it's always located on the first physical sector of the floppy. With a hard disk, the boot record is located on the first logical sector of each partition, of which there may be more than one. The first physical sector of a hard disk contains the master boot record, which defines the partition layout for the remainder of the disk.

With this in view, take a look at Listing 4. The routine shown here, GetMediaType, simply reads the first sector from a disk and compares the bsSectors field of the boot record against the corresponding entries in the DiskTypes array. If a match is found, then the media type is returned as an integer into this array. If no match is found, the value DF_Unknown is returned (one of those ancient disk formats we mentioned earlier!) and if the boot sector can't be read, then we return -1. This routine will be used later in the Quick Format logic.

```
function AbsRead (Drive: Byte; NumSectors, StartSec: Integer;
  Buffer: Pointer): Integer; assembler;
asm
  mov     l,Drive    { AL = drive number for read  }
  dec     al         { for compatability, A=1,...  }
  mov     cx,NumSectors    { number of sectors to read }
  mov     dx,StartSec      { first sector to read }
  push    ds       { save DS prior to call }
  lds     bx,Buffer    { DS:BX = pointer to buffer }
  push    bp       { save stack frame }
  int     25h      { do the absolute read }
  pop     bx       { pop and discard flags }
  pop     bp       { restore stack frame }
  pop     ds       { restore DS register }
  jc     @@1       { branch if error }
  xor     ax,ax    { no error - return zero }
@1:
end;
function AbsWrite (Drive: Byte; NumSectors, StartSec: Integer;
  Buffer: Pointer): Integer; assembler;
asm
  mov    al,Drive    { AL = drive number for write }
  dec    al      { for compatability, A=1,... }
  mov    cx,NumSectors  { number of sectors to write }
  mov    dx,StartSec    { first sector to write }
  push   ds      { save DS prior to call }
  lds    bx,Buffer    { DS:BX = pointer to buffer }
  push   bp      { save stack frame }
  int    26h     { do the absolute write }
  pop    bx      { pop and discard flags }
  pop    bp      { restore stack frame }
  pop    ds      { restore DS register }
  jc    @@1      { branch if error }
  xor    ax,ax   { no error - return zero }
@1:
end;
```

➤ *Listing 2*

```
type
  PBootSector = ^BootSector;
  BootSector = record
    bsJump: array [0..2] of Byte;    { 00 E9 XX XX or EB XX 90             }
    bsOemName: array [0..7] of Char; { 03 OEM name and version             }
    bsBytesPerSec: Integer;          { 0b bytes per sector                 }
    bsSecPerClust: Byte;             { 0d sectors per cluster              }
    bsResSectors: Integer;           { 0e number of reserved sectors       }
    bsFATs: Byte;                    { 10 number of file allocation tables }
    bsRootDirEnts: Integer;          { 11 number of root-directory entries }
    bsSectors: Integer;              { 13 total number of sectors          }
    bsMedia: Byte;                   { 15 media descriptor                 }
    bsFATsecs: Integer;              { 16 number of sectors per FAT        }
    bsSecPerTrack: Integer;          { 18 sectors per track                }
    bsHeads: Integer;                { 1a number of heads                  }
    bsHiddenSecs: LongInt;           { 1c number of hidden sectors         }
    bsHugeSectors: LongInt;        { 20 number of sectors if bsSectors = 0 }
    bsDriveNumber: Byte;             { 24 drive number                     }
    bsReserved1: Byte;               { 25 reserved                         }
    bsBootSignature: Byte;           { 26 extended boot signature          }
    bsVolumeID: LongInt;             { 27 volume ID number                 }
    bsVolumeLabel: array [0..10] of Char; { 2b volume label                }
    bsFileSysType: array [0..7] of Char;  { 36 file-system type            }
  end;
```

➤ *Listing 3*

Lock Up Your Volumes!

There's another important consideration which Microsoft introduced along with Windows 95. Windows has always been a multitasking system, but it's only with the advent of Windows 95 that it becomes pre-emptive to any real extent (NT has always been truly pre-emptive, but that's another story...). Prior to Windows 95, applications only yielded control to other running programs when they executed certain critical API

```
function GetMediaType (Drive: Byte): Integer;
var
  i: Integer;
  buff: array [0..511] of Byte;
  bs: BootSector absolute buff;
begin
  GetMediaType := -1;
  { Read boot sector from disk }
  if AbsRead (Drive, 1, 0, @buff) = 0 then begin
    GetMediaType := DF_Unknown;
    for i := DF_360K to DF_28M do
      if bs.bsSectors = DiskTypes [i].sec then begin
        GetMediaType := i;
        Exit;
      end;
  end;
end;
```

➤ *Listing 4*

```
function LUVolumePrim (Drive, Level, Op: Byte; Perm: Word): Integer;
  assembler;
asm
  mov   ax,$440D      { specify generic IOCTL call }
  mov   bl,Drive      { get drive number in BL     }
  dec   bl            { for compatability, A: = 1  }
  mov   bh,Level      { get lock level in BH       }
  mov   ch,8          { category 8 for drives      }
  mov   cl,Op         { get lock/unlock physical   }
  mov   dx,Perm       { get permissions word       }
  int   21h           { make the call              }
  jc    @@1           { branch if error            }
  xor   ax,ax         { no error - so AX = 0       }
@1:
end;

function LockVolume (Drive: Byte): Integer;
begin
  if IsWindows95 then begin
    LockVolume := -1;
    if LUVolumePrim (Drive, 0, $4B, 0) = 0 then
      if LUVolumePrim (Drive, 0, $4B, 4) = 0 then
        LockVolume := 0
  end else
    LockVolume := 0;
end;

function UnLockVolume (Drive: Byte): Integer;
begin
  if IsWindows95 then begin
    UnLockVolume := -1;
    if LUVolumePrim (Drive, 0, $6B, 0) = 0 then
      if LUVolumePrim (Drive, 0, $6B, 0) = 0 then
        UnLockVolume := 0;
  end else
    UnLockVolume := 0;
end;
```

➤ *Listing 5*

calls such as `GetMessage` and `PeekMessage`. Now however, an application can be pre-empted at any time.

What happens if application A is formatting a floppy disk and then application B comes along and tries to access the disk? The result can only be trouble of one sort or another! In order to address this sort of problem, Microsoft introduced the concept of volume locking. When an application wants to format a disk, it must obtain a lock for that volume. If it doesn't have the lock, Windows will "bounce" certain operating system calls including track formatting and absolute sector writes. At the same time, an application which doesn't have the lock must clearly be prevented from accessing a locked volume.

In reality, things are actually quite a bit more complex than this. Microsoft have implemented a multi-level locking system with four different levels of lock. As you move up through the locking levels from 0 to 3, things become progressively more restrictive for those applications which don't hold the lock. Just to make things a little more complex, it's actually the least restrictive lock (level zero) which has a special, super restrictive mode that's used only when formatting disks. Confused? You certainly will be...

Microsoft's recommendation is that a disk-formatting application should first obtain a standard level 0 lock. It can then lock the volume a second time to obtain the more restrictive lock. This is done by specifying a value of 4 for the associated permissions byte. While this lock is held, the disk can be formatted and the program must ensure that it does not release the innermost, restrictive, level 0 lock until the disk has been put into a state where it can be recognised as a normal, FAT-based volume. Finally, the outermost level 0 lock can be released.

Listing 5 contains two routines, `LockVolume` and `UnLockVolume`, which encapsulate my interpretation of Microsoft's algorithm for locking and unlocking volumes. They both use a lower-level routine which is not exported from the interface part of the unit. Because the volume locking functionality requires Windows 95, these routines make a special check that they're running under Windows 95, or a later version thereof. The code to do this is included as part of the unit, FORMAT.PAS, which is included on this month's disk.

Coming Soon...

This time round there is no sample program to play with: the FORMAT unit is simply laying the groundwork for the disk formatting code proper which I'll present next month, so you'll have to be patient for a little longer! The DISKINFO unit remains the same.

When not sticking pins in the wax effigies of Microsoft system programmers, Dave Jewell is writing a new book on 32-bit Delphi and the Windows API, due to be published around the middle of the year, by Wrox Press. You can contact Dave on CIX as djewell@cix.compulink.co.uk, on CompuServe as 102354,1572 or as DSJewell on America OnLine.