Delphi Internals: How not to Write an Operating System (4)

Disk formatting – nearly there but not quite!

by Dave Jewell

Or: A report on the correlation between writing articles on disk formatting and suicide amongst computer journalists...

For all of us, there are certain memorable moments in our lives where, with the benefit of hindsight, we fervently wish that we'd done things differently. One such moment came when I stupidly agreed to write a few articles on the subject of floppy disk formatting. In the words of Proverbs 7 verse 22, I was *"like an ox going to the slaughter..."* Little did I know the blood, sweat and tears that would accompany this seemingly innocent subject.

The fact is that disk formatting is something of a minefield. In the course of writing this month's article, I must have re-written the code half a dozen times in an attempt to produce something that would work on all three of the machines I routinely use. I have aged ten years in the last couple of weeks, my hands tremble slightly at the keyboard and sometimes I wake up screaming in the middle of the night...

Does this sound like a thinly-veiled attempt to justify the late delivery of this missive into the eager hands of our esteemed Editor? Well, yes, there's probably an element of self-justification here, but it's nevertheless true that if you want to create a disk formatting routine that will work all the time on lots of different machines, with a large number of different BIOS manufacturers, then quite a bit of experimentation is required. To make it all that much more difficult, creating code that works in DOS is one thing, but to get something that works properly in Windows is *quite* another.

This month's article presents the code to format the tracks of a floppy. In addition, the code builds a boot block and writes this to the disk. In next month's (hopefully final) article on disk formatting, I'll finish with the code that writes an initial FAT (File Allocation Table) to the disk and then wrap the whole thing up into a Delphi program with a nice user interface.

Media Sensing Revisited

Last month I introduced a routine, `GetMediaType`, to sense the type of diskette inserted into a drive. A 5.25 inch drive can support 360Kb and 1.2Mb floppies (remember, we decided last time not to support the really old formats) whereas a 3.5 inch drive supports 720Kb, 1.44Mb and 2.88Mb disks. The purpose of the `GetMediaType` routine was to find which type of disk was in the drive, to be used when quick formatting a disk.

Well, the bad news is that this routine is now defunct. It's been replaced with `SenseMediaType` (see Listing 1). This procedure is more flexible than its predecessor because it takes a flag, `fDefault`, that determines whether we're talking about the disk installed in the drive or the default capabilities of the drive. Let me explain: suppose you stick a 1.44Mb disk into drive A: and tell our not-yet-finished program that you want to quick-format it. The program has to read the disk to determine its current capacity so that it can then set up a new BPB (Bios Parameter Block) and empty FATs which agree with the capacity of the disk. In order to do this, you'd call the `SenseMediaType` routine with `fDefault` set to `False` – you don't want the default state of play, you're interested in the disk that's physically in the drive now.

But suppose the disk is damaged (after all, this might be why we're

➤ *Listing 1*

```
function SenseMediaType(Drive: Byte; fDefault: Bool; pdp: PDeviceParams):
  Integer;
var count: Integer;
begin
  FillChar (pdp^, sizeof (DeviceParams), 0);
  if not fDefault then
    pdp^.SpecFunc := 1;
  asm
    mov     ax,440Dh        { Specify generic IOCTL call     }
    mov     bl,Drive        { BL = wanted drive number       }
    mov     cx,$0860        { Request device parameters      }
    push    ds              { save DS register               }
    lds     dx,pdp          { ds:dx points to param block    }
    call    Dos3Call        { make the call                  }
    pop     ds              { restore DS register            }
    jc      @@1             { if error, return code in AX    }
    xor     ax,ax           { else clear AX register         }
  @1:
    mov     count,ax        { stash result in 'err'          }
  end;
  if count <> 0 then
    SenseMediaType := -1
  else for count := DF_360K to DF_28M do
    if pdp^.bpb.bsSectors = DiskTypes [count].sec then begin
      SenseMediaType := count;
      Exit;
    end;
end;
```

```
PDeviceParams = ^DeviceParams;
DeviceParams = record
  SpecFunc: Byte;                  { 00 }
  DevType: Byte;                   { 01 }
  DevAttrs: Integer;               { 02 }
  Tracks: Integer;                 { 04 }
  MediaType: Byte;                 { 06 }
  bpb: BPB;                        { 07 }
  bsHidden2: Integer;              { 1A }
  HugeSectors: LongInt;            { 1C }
  Reserved: array [0..5] of Char;  { 20 !! UNDOCUMENTED !! }
  { Start of TRACKLAYOUT information }
  SectorsPerTrack: Integer;        { 26 }
  TrackLayout: array [0..35] of LongInt;  { 28 }
end;
```

➤ *Listing 2*

reformatting it) or suppose you haven't even put it in the drive yet? In this case, the above call to `SenseMediaType` will fail and we'll have to display a message stating that the disk can't be quick formatted. Because the punter (a technical term relating to the user of our program!) has said that he wants to quick-format the disk, he may not have specified what disk capacity he wants. Consequently, the program has to ask DOS for details of the default capacity of the drive – that's what happens when you set `fDefault` to `True`. No disk has to be in the drive, it's purely an internal DOS call which doesn't "hit" the disk at all.

`SenseMediaType` also takes a third parameter, a pointer to a new structure called `DeviceParams`. The layout of this data structure is shown in Listing 2. The figures in comments are hexadecimal offsets into the structure.

The `SpecFunc` field is used to control whether we're getting the default device parameter block for a drive, or whether we're examining the currently inserted disk. This field is set up from the `fDefault` parameter. Strictly speaking, the `SenseMediaType` routine only needs to return an integer indicating the capacity of the specified drive, but (as we shall see later) there are other reasons for needing access to the device parameter block, which is why the data structure is passed.

I will explain the other bits of this data structure where relevant as the rest of this month's code is discussed but, in passing, you'll also notice that there's an undocumented six-byte area at offset `$20`. I have nowhere seen this mentioned in the Microsoft documentation and it was only through trial and error, combined with some intensive scrutiny of the Windows 3.1 File Manager code, that I was able to infer the existence of this area. But then, that's par for the course as far as these low-level routines are concerned. The `TrackLayout` information is very important and is described later.

Introducing FormatDisk

The code in Listing 3 is the heart of the `FormatDisk` routine itself. This is a relatively high-level routine which will ultimately be called directly from the Delphi application. It takes a drive specifier (1 for drive A:, 2 for drive B:) and a `Size` parameter which is an index into the `DiskTypes` array that we looked at last time.

Incidentally, the `DiskTypes` array has also changed slightly from last month. It now includes an extra field which gives the total number of clusters for each possible disk capacity. If you're interested in playing around with this code, let me strongly emphasise that you should work with the code *only* from one specific month's disk. Don't try and mix this month's code with last month's or bad things might happen. If you end up formatting your hard disk then remember that you were warned!

This being the case, the first thing the `FormatDisk` routine does is check that the drive parameter is either 1 or 2. Most of the DOS calls I'm making simply won't work with hard disks, but it's better to be safe than sorry. There's also a new global variable, `fAbort`, to be used by the host program to tell the disk formatting code to stop. Typically, you'd use this to implement a `Cancel` button while the actual formatting is in progress. In the final version of the code, this variable will be exported through the `interface` part of the unit so that it's accessible to the host code.

Next, we call `SenseMediaType` to get a device parameter block for the required drive. The resulting DPB (let's call it that to be concise) is stored in the `OldDeviceParams` global. This is a very important step, up to this point you might have been thinking 'Who cares about DPBs and what are they for?' The point is that DOS uses DPBs to "switch" a floppy disk drive to a particular capacity. If we wanted to format a 720Kb disk in a 1.44Mb drive then we first need to set the drive to that capacity. Similarly, if you wanted to format a 360Kb disk in a 1.2Mb drive then you'd need to switch to that format. Of course, if a particular drive is already "switched" to the required format then you don't need to change it. What is important, though, is to save the current drive configuration and restore it at the end of the format process. That's why we must store a copy of the drive's current DPB in `OldDeviceParams`.

It's possible, incidentally, to set up invalid combinations of drive and capacity. For example, you can set a 1.44Mb drive to format 360Kb floppy disks – DOS won't object, but you might end up with a strange diskette! In order to prevent this happening, the `FormatDisk` routine assumes that the higher-level code will only request valid drive/size combinations but, if you wanted, you could use the drive detection code from previous articles to put explicit checks into the `FormatDisk` routine itself.

There are other ways of accomplishing the same task. At the ROM BIOS level, there are calls for setting floppy drives to a particular capacity. However, from what I've seen of Microsoft's code, it's best to avoid the BIOS routines if at all possible.

```
function FormatDisk(Drive, Size: Integer): Integer;
label
  Stop;
var
  pDisk: PDiskType;
  TracksLeft, TotTracks, CurTrk, CurHead,
  CurSector,err, Cluster, SysSectors, DefSize,
  DiskSize, count: Integer;
begin
  { Assume failure and validate drive number }
  FormatDisk := -1;
  fAbort := False;
  if not Drive in [1..2] then
    Exit;
  { Stash current drive setup }
  DefSize := SenseMediaType(
    Drive, True, @OldDeviceParams);
  { If we're quick-formatting, then auto-sense
    the current media }
  DiskSize := Size;
  if DiskSize = -1 then
    DiskSize := SenseMediaType (Drive, False, @dp);
  { If media not present or other error, slow-format }
  if DiskSize = -1 then begin
    if MessageDlg ('Can''t quick-format this disk.'+
      ' Format to default capacity?', mtConfirmation,
      [mbYes, mbNo], 0) = mrNo then
      Exit;
    DiskSize := DefSize;
  end;
  { Establish wanted media size with DOS }
  if DiskSize <> DefSize then
    if SetMediaType (Drive, DiskSize) <> 0 then
      Exit;
  { Grab disk params table }
  pDisk := @DiskTypes [DiskSize];
  FormatInit;
  { Tweak disk params table for wanted format }
  DiskParams [4] := Chr (pDisk^.spt);
  if pDisk^.spt = 15 then
    DiskParams [7] := Chr ($54)
  else
    DiskParams [7] := Chr ($50);
  { Now we can format the tracks }
  if DiskSize = 0 then
    TotTracks := 80
  else
    TotTracks := 160;  { Heads=2! }
  SysSectors :=
    (2*pDisk^.spf)+(((pDisk^.rde*32)+511)div 512)+1;
  TracksLeft := TotTracks;
  CurHead := 0;
  CurTrk := 0;
  { Only format tracks if not quick formatting }
  if Size <> -1 then begin
    { Main formatting loop }
    while TracksLeft <> 0 do begin
      { Let somebody else get a look-in! }
      Application.ProcessMessages;
      if fAbort then
        goto Stop;
      if FormatTrack (Drive, CurTrk, CurHead) = -1 then
        goto Stop;
      CurSector := ((CurTrk*2)+CurHead)*pDisk^.spt;
      count := CurSector;
      while count < CurSector + pDisk^.spt do begin
        Cluster := ((count-SysSectors) div pDisk^.spc)+2;
        FatMask[Cluster shr 3] := FatMask[Cluster shr 3]
          or (1 shl (Cluster and 7));
        Inc (count);
      end;
      Dec(TracksLeft);
      Inc(CurHead);
      if CurHead >= 2 then begin
        CurHead := 0;
        Inc (CurTrk);
      end;
    end;
  end;
  { Write a new boot sector to the disk }
  WriteBootSector (Drive, @TargetBPB);
  { Let somebody else get a look-in! }
  Application.ProcessMessages;
  if fAbort then goto Stop;
  { !!! WATCH THIS SPACE !!! }
  { ... }
Stop:
  SetMediaType (Drive, -1);
  FormatTerminate;
end;
```

➤ *Listing 3*

If you want to quick-format a disk, then you pass a value of -1 as the `Size` parameter to `FormatDisk`. In this case, `SenseMediaType` is used to determine the currently inserted media type. If this routine fails, (disk not inserted or completely mangled) then the code offers to format the disk at the default capacity for that drive.

The next job is to call the `SetMediaType` routine (Listing 4). Using the default DPB block as a starting point, it attempts to build a new DPB which matches the required capacity of the drive. In order to do this, it primarily uses the condensed BPB information contained in the `DiskTypes` array. You'll notice that if a value of -1 is passed for the `Size` parameter it tells the routine to switch the drive back to the default DPB. It's called in this way right at the end of the `FormatDisk` code.

There's another important job which `SetMediaType` has to perform. As we've already seen, there's a special 'track layout' area at the end of a DPB. This is used to tell DOS the size of each sector and the order in which the individual sector address marks are written to the disk. In practice, disk sectors are always the same size (512 bytes) and they're arranged consecutively around the track.

Going back to the `FormatDisk` routine, the procedure `FormatInit` is called next. Having said that we're trying to avoid BIOS issues wherever possible, for the sake of code portability across different machines, there are some circumstances where this isn't possible. To put things another way, there are some BIOSs which are so dumb that they need a bit of a helping hand. The software interrupt vector `$1E` isn't really an interrupt at all. Rather, it's a pointer to a special memory area called the disk parameter table. This contains low-level information that's used by your PC's ROM BIOS to access the floppy disk and, in particular, to format disks. You might reasonably expect that this table would be set up automatically by DOS when we call `SetMediaType`, but meanwhile, back in the real world...

The `FormatInit` routine makes a copy of the disk parameter table in a global variable and sets up a pointer to the 'live' parameter table by establishing a pointer, `DiskParams`. By making this pointer of type `PChar` we can easily access the disk parameter table as though it were an ordinary array declared inside our program. There are two bytes which have to be tweaked: the byte at offset 4 (fifth byte along), which determines the number of sectors per track, and the byte at offset 7, which tells the BIOS how much of an inter-sector gap to place between sectors when formatting the disk.

```
function SetMediaType (Drive, Size: Integer): Integer;
var
  err: Byte;
  p: Pointer;
  sec: Integer;
  dp: DeviceParams;
begin
  { Use default diskparams as starting point }
  dp := OldDeviceParams;
  if Size = -1 then
    dp.SpecFunc := 4
  else begin
    { Set up 'dp' according to wanted disk size }
    dp.SpecFunc := 5;
    dp.DevType := Size;
    if Size = 3 then dp.DevType := 7;
    if Size = 4 then dp.DevType := 9;
    if Size = 0 then begin
      dp.Tracks := 40;
      dp.MediaType := 1;
    end;
    dp.bpb.bsBytesPerSec := 512;
    dp.bpb.bsSecPerClust := DiskTypes [Size].spc;
    dp.bpb.bsResSectors  := 1;
    dp.bpb.bsFATs        := 2;
    dp.bpb.bsRootDirEnts := DiskTypes [Size].rde;
    dp.bpb.bsSectors     := DiskTypes [Size].sec;
    dp.bpb.bsMedia       := DiskTypes [Size].med;
    dp.bpb.bsFATsecs     := DiskTypes [Size].spf;
    dp.bpb.bsSecPerTrack := DiskTypes [Size].spt;
    dp.bpb.bsHeads       := 2;
    dp.bpb.bsHidden1     := 0;
    TargetBPB := dp.bpb;
    dp.bsHidden2 := 0;
    dp.HugeSectors := 0;
    dp.SectorsPerTrack := dp.bpb.bsSecPerTrack;
    for sec := 0 to dp.SectorsPerTrack - 1 do
      dp.TrackLayout [sec] := MakeLong (sec + 1, 512);
  end;
  { Now tell DOS this is what we want! }
  p := @dp;
  err := 0;
  asm
    mov  ax,$440D          { specify generic IOCTL call }
    mov  bl,byte ptr Drive { BL = drive number          }
    mov  cx,$0840          { set device parameters      }
    push ds                { save DS on stack           }
    lds  dx,p              { get pointer to ParamBlock  }
    call DOS3Call          { do the business...         }
    pop  ds                { restore DS register        }
    jnc  @@1               { branch if no error         }
    mov  err,ah            { stash error code           }
  @1:
  end;
  SetMediaType := err;
end;
```

➤ *Listing 4*

Track Formatting

At this point, we're all set to format the track information onto the disk. `TotTracks` is set to the total number of tracks, allowing for the fact that we've got tracks on both sides of a double-sided disk! It's used by the main track formatting loop to determine when we're done.

You'll notice that track formatting is omitted if quick-formatting is selected (`Size` parameter is -1). Otherwise, the code iterates along, formatting a total of 80 tracks on a double-sided 40 track disk, or 160 tracks on a double-sided 80 track disk. Immediately before each track is formatted, we call the VCL's `Application.ProcessMessages` method which allows other Windows applications to get a look in. Under Windows 3.1, failing to do this would lock out everything else while the formatting operation was in progress. Calling `Application.ProcessMessages` also allows the punter to press a cancel button, setting the `fAbort` variable. This is also polled once around each loop.

For the sake of space, I haven't provided a listing for the `FormatTrack` routine, suffice to say that it uses another IOCTL call to format a disk track. All the code discussed this month is on the disk anyway. The `FatMask` array is used to set up an array of cluster masks. This in turn is used to write the initial FAT tables to the floppy disk – I'll be covering that next month.

If you're concerned about maximum safety, you could easily modify the track formatting code so that it started with the innermost tracks and worked its way outwards. The advantage of this is that the most important parts of a disk (the BPB, FATs and directory information) are located on the first few tracks. If you ever accidentally start formatting a disk and suddenly realise that your business data is going up in smoke, formatting from the outermost tracks last will generally enable you to recover much of your data provided that the formatting process hasn't proceeded too far.

It's also nice, in any decent formatting program, to provide some visual indication of how far along the formatting process is. I'll be adding an application-supplied progress hook next month.

Once the tracks have been formatted, the final job, for now, is to call the `WriteBootSector` routine. As the name suggests, the purpose of this code is to create a valid boot sector image and write it to the beginning of the disk. The boot sector image contains a BPB, the serial number of the disk and a certain amount of machine code which tries to load the IO.SYS and MSDOS.SYS files into memory. For reasons of simplicity (this thing is taking far too long as it is!) I made the decision not to support the creation of bootable, system disks, but the boot sector image code is valid in case you want to add this feature.

The boot sector image is contained in the global `FloppyBoot` array. This is a valid image taken from an existing 1.44Mb diskette. The `WriteBootSector` code takes this image, adds in an appropriate BPB (this is why the `SetMediaType` routine stashes away a copy of the newly-created BPB in the `TargetBPB` variable, so that we can re-use it here) and then calls the `GenSerialNumber` routine to dream up an acceptable serial number for the disk. As you can see, this code simply combines the current date and time into a 32-bit number that constitutes the disk serial number. Since this includes a 1/100th second count, there's no possibility of winding up with two diskettes with the same serial number unless you format them on two machines simultaneously or start fiddling around with your PC's date/time settings. The algorithm used here is the same as that used by the Windows File Manager. When Microsoft first added serial numbering to floppy disks there was considerable debate in the on-line programming community as to

```
function GenSerialNumber: LongInt; assembler;
asm
  mov     ah,$2A              { request system date }
  call    DOS3Call            { result in CX:DX     }
  push    cx                  { push year part      }
  push    dx                  { push month/day      }
  mov     ah,$2C              { request system time }
  call    DOS3Call            { result in CX:DX     }
  pop     ax                  { pop month/day       }
  add     ax,dx               { add to seconds/100  }
  pop     dx                  { pop year part       }
  add     dx,cx               { add hours/minutes   }
end;
function WriteBootSector (Drive: Byte; SrcBPB: PBPB): Integer;
const
  BPBSig: array [0..7] of Char = 'FAT16   ';
var
  DestBPB: PBPB;
  i: Integer;
  SerNum: LongInt;
  BootSector: array [0..511] of Byte;
begin
  { Get a copy of the default boot record }
  Move (FloppyBoot, BootSector, sizeof (BootSector));
  { Add the BPB for this specific disk capacity }
  DestBPB := @BootSector [11];
  DestBPB^ := SrcBPB^;
  { Init extended boot stuff }
  for i := $1E to $24 do
    BootSector [i] := 0;
  SerNum := GenSerialNumber;
  Move (SerNum, BootSector [$27], sizeof (SerNum));
  Move (BPBSig, BootSector [$36], 8);
  WriteBootSector := WriteAbs (@BootSector, Drive, 0, 0);
end;
```

➤ *Listing 5*

what algorithm was being used. Well, here it is in all its glory – not really much to it, is there?

Well, that's it for this month. Next month, it's my intention either to finish this disk formatting saga or else ritually disembowel myself. And no, you don't get to express a preference!

When not sharpening his ceremonial Japanese disembowelling sword, Dave Jewell is writing a new book on 32-bit Delphi and the Windows API. If he lives that long, it's due to be published around the middle of the year by Wrox Press. If you want to send flowers to his widow, she will be monitoring Dave's email on CIX as djewell@cix.compulink.co.uk, on CompuServe as 102354,1572 or as DSJewell on America OnLine.