Delphi Internals: How Not To Write An Operating System (2)

Drive serial numbers and CD-ROM detection

by Dave Jewell

Last month, as you will no doubt remember, I presented a number of low-level routines which reported the number and type of the floppy disk drives attached to your PC and enabled you to read and write volume labels from within a Delphi application. Well, they say that a week is a long time in politics and the same is certainly true of a month in computer journalism! Not being one to tolerate slackers, the editor spotted the fact that I'd bottled out of the serial number issue and sent me some public-domain code which reads and writes serial numbers to DOS volumes. OK, I can take a hint...

Serial Number Handling

Microsoft implemented serial numbers in DOS 4.0 onwards partly as an aid to those who wanted to protect commercial software against illicit copying. The idea is that the DOS `FORMAT` command marks each DOS volume with a 32-bit serial number when the volume is created. Microsoft have never published the details of how the 32-bit serial number is generated, but I believe that the current date and time has some bearing.

The routines described here will allow you to change the serial number of any DOS volume, including a hard disk. I would advise you to exercise some caution here since if you have installed a piece of software that checks serial numbers (and some anti-piracy techniques work just this way), it won't take kindly to finding a different number to what it saw last time round. It will assume that it's been moved to a different hard disk and refuse to play ball. You have been warned... As I said last month, the ultimate aim of this mini-series on low-level DOS hackery is to come up with a Delphi program that allows you to format and copy DOS floppies. As far as serial number twiddling is concerned, I'd leave the hard disk serial number alone if I were you.

Like the volume label information, a disk's serial number is part of the `MIDINFO` data structure that I described last month. Unfortunately, because of the way that DOS and Windows inter-relate, it isn't possible to simply issue a DOS call to retrieve the required serial number information.

As I explained, Windows intercepts any DOS call and tries to handle the call itself. This is done as a performance optimisation and saves having to switch the processor down to real mode before calling *Ye Olde Real-Mode DOS Kernel*. Now the problem here is that, under Windows 3.1, there are a fair number of calls that Windows doesn't know how to handle: it just passes them directly down to real-mode DOS. As luck would have it, the DOS handler in Windows doesn't understand any of the calls that relate to volume serial numbers.

Just to make matters worse, all the calls relating to volume serial numbers rely on passing the address of a `MIDINFO` data structure to DOS. This address is passed in the `DS:DX` registers. Now, since the DOS handling code in Windows doesn't understand these DOS calls, it can't possibly know the significance of the `DS` and `DX` registers, it simply passed them unchanged to the real-mode kernel code. The upshot of this is that the real-mode kernel gets handed a protected mode address which it can't make use of. It's not surprising, therefore, that the DOS calls in question don't work.

DPMI To The Rescue...

Fortunately, the DPMI extender built into Windows provides a solution. If you're not familiar with DPMI (DOS Protected Mode Interface), suffice it to say that it's a set of low-level operating system services that are designed to prevent DOS extenders from treading on one another's toes.

You might think that Windows itself is responsible for organising the memory in your PC, but you'd be wrong – the overall responsibility belongs to whatever DOS extender is loaded. Normally, this is the DOS extender that's loaded as part of the Windows initialisation, it's responsible for handing out memory to Windows and to any DOS sessions that are started up. It also allocates memory to protected mode DOS extenders which may start running within a DOS session.

Besides memory management, the DOS extender also contains routines which enable protected mode programs to call real-mode interrupts. Aha! This is just what we want to hear. By getting our Windows app to call the real-mode DOS `$21` interrupt directly, we can bypass the brain-dead DOS code in Windows and communicate directly with the DOS kernel.

This leaves just one problem. We need to ensure that the `MIDINFO` data structure is allocated in a place where it can be accessed both by a protected mode application and also by the real-mode DOS kernel. As you may remember from last time, even the mere act of using the Pascal `INTR` routine can cause a GPF here by virtue of an invalid value being loaded into the processor's registers.

Once again, there's a way around this problem. Windows provides a special routine, `GlobalDOSAlloc`,

which guarantees to allocate memory in the first Mb of RAM. Such memory is available both to real and protected-mode programs. For this reason, it's a very scarce and precious system resource. You must be careful to allocate only the smallest necessary amount of memory using this technique and to de-allocate it again as soon as possible.

Putting all this together, I came up with the SERNUM unit which appears in Listing 1. As a starting point, I based this unit on code from a public-domain application note published by Borland (number 2534) but the code is now about half the size it was originally, is much more structured and has a far simpler and cleaner interface. For example, 'clients' of this unit do not need to know about the MIDINFO data structure, the use of DPMI or the vagaries of GlobalDOSAlloc!

All the serial number twiddling code is shown here as a separate unit for the sake of convenience and for conciseness since I didn't want to replicate any of the code from last month. However, to make life easier for you and me, I also added the code to the DOSINFO unit which we began last month – the new version is on this month's disk of course.

How It Works

The SERNUM unit exports just two routines: GetSerialNumber and SetSerialNumber. The code in the GetSerialNumber routine simply calls the lower-level GetMid routine to fetch the MIDINFO data structure for the specified drive. The volume serial number is then extracted from this and returned as the function result. The SetSerialNumber routine is equally straightforward: it calls GetMid to retrieve the current MIDINFO record, modifies just the serial number field and then writes the record back to disk through a call to SetMid.

At the next level of implementation, the GetMid and SetMid routines both call GlobalDOSAlloc to allocate space for a MIDINFO data structure in the first Mb of RAM. This is for the reasons given earlier. The function result from GlobalDOSAlloc is a 32-bit value which contains a protected mode selector in the low-order word and a real mode segment address in the high-order word. It is absolutely crucial that the protected mode Windows

➤ *Listing 1*

```
unit SerNum;
Interface
uses WinProcs, WinTypes;
function GetSerialNumber(drive: Byte): LongInt;
function SetSerialNumber(drive: Byte; serNum: LongInt):
  Bool;
Implementation
type
  PMIDINFO = ^MIDINFO;
  MIDINFO = record
              InfoLevel:  Word;
              SerialNum:  Longint;
              VolLabel:   array[0..10] of Char;
              FileSystem: array [0..7] of Char;
            end;
var
  R: record     { Real mode call structure }
       di, si, bp, Reserved, bx, dx, cx, ax : Longint;
       Flags, es, ds, fs, gs, ip, sp, ss: Word;
     end;
function GetSetMid(Drive: Byte; MID: PMIDINFO;
  RealModeAX: Word): Bool;
{ Low level code to get or set a MIDINFO data structure
  for the specified drive; RealModeAX = $6900 for a get
  and $6901 for a set operation }
var Error: Byte;
begin
  Error := 0;   { Assume everything ok }
  GetSetMid := True;
  R.ax := RealModeAX;
  R.bx := Drive;
  R.ds := HiWord(Longint(MID));       { Subtle !!! }
  R.dx := LoWord(Longint(MID));
  asm
    mov bx, 0021h      { set flags to $00, Real mode interrupt $21 }
    xor cx, cx         { copy 0 words from protected mode stack }
    mov ax, seg R
    mov es, ax         { selector of real mode call structure }
    mov di, offset R   { offset of real mode call structure }
    mov ax, 0300h      { DPMI simulate real mode interrupt }
    int 31h            { do the business }
    jnc @@1            { branch if no error }
    inc Error
    @@1:
  end;
  if Error = 1 then GetSetMid := False;
end;
function GetMid(drive: Byte; var mid: MIDINFO): Bool;
{ Get the MIDINFO record for a specified drive, uses
  GetSetMid, returns TRUE if successful }
var p: LongInt;
begin
  GetMid := False;  { Assume failure }
  { Allocate a MIDINFO data structure in DOS address-space }
  p := GlobalDOSAlloc(sizeof(MIDINFO));
  if GetSetMid(drive, Ptr(HiWord(p), 0), $6900) then
  begin
    mid := PMIDINFO(Ptr(LoWord(p), 0))^;
    GetMid := True;
  end;
  GlobalDOSFree(LoWord(p));
end;
function SetMid(drive: Byte; var mid: MIDINFO): Bool;
{ Set the MIDINFO record for a specified drive, uses
  GetSetMid, returns TRUE if successful }
var p: LongInt;
begin
  SetMid := False;   { Assume failure }
  { Allocate a MIDINFO data structure in DOS address-space }
  p := GlobalDOSAlloc(sizeof(MIDINFO));
  PMIDINFO(Ptr(LoWord(p), 0))^ := mid;
  if GetSetMid(drive, Ptr(HiWord(p), 0), $6901) then
    SetMid := True;
  GlobalDOSFree(LoWord(p));
end;
function GetSerialNumber(drive: Byte): LongInt;
{ Get the serial number for a specified drive, if an
  error occurs, then 0 is returned as the serial number }
var mid: MIDINFO;
begin
  if GetMid(drive, mid) then
    GetSerialNumber := mid.SerialNum
  else
    GetSerialNumber := 0;
end;
function SetSerialNumber(drive: Byte; serNum: LongInt):
  Bool;
{ Set the serial number for a specified drive, if no
  error, TRUE is returned as the function result }
var mid: MIDINFO;
begin
  SetSerialNumber := False;
  if GetMid(drive, mid) then begin
    mid.SerialNum := serNum;
    SetSerialNumber := SetMid(drive, mid);
  end;
end;
end.
```

program only accesses the MIDINFO data structure using the selector. If the real mode segment gets inadvertently loaded into a segment register, you'll get an immediate GPF error. It is remarkably easy to do this unintentionally, as we'll see in just a moment.

Having allocated memory in DOS's address space, The GetMid routine calls GetSetMid, passing it a real-mode pointer to the MIDINFO buffer and specifying a DOS function code of $6900. This tells DOS to read the serial number information from the disk. Similarly, the SetMid routine allocates a MIDINFO structure, copies the passed data into it (so that it can be accessed by DOS) and calls GetSetMid with a DOS function code of $6901 indicating a write operation.

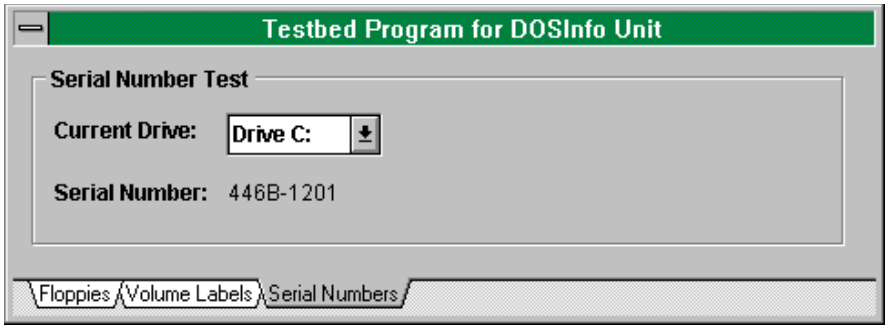
The real low-level magic, of course, takes place in the GetSetMid function. In the original Borland technical note, this job was done by two routines which differed only by one line. I made things a lot simpler by just passing the required DOS function code as the RealModeAX parameter.

The first job is to initialise the various fields of the R variable according to the register values that we want the real-mode DOS kernel to see. As far as the $6900/$6901 routines are concerned, only the AX, BX and DS:DX registers are relevant here.

Now, here's a little subtlety for you: look closely at the two assignment statements where the R.ds and R.dx fields are initialised. If you've much familiarity with Borland's Pascal dialect, you'll immediately spot the fact that these statements don't look very efficient. It would be more efficient to use the built-in Ofs and Seg operators like this:

```
R.ds := Seg(MID^);
R.dx := Ofs(MID^);
```

The trouble with is that these statements will immediately produce – yes, you've guessed – another GPF. Although it might seem counter-intuitive, the mere use of the Seg and Ofs operators will load the ES segment register with the specified



➤ *Here's last month's testbed program modified to display serial numbers. It would be dead easy to get it to also change serial numbers, but you'll have to take responsibility for this yourself!*

pointer. Since it's a real-mode rather than a protected-mode pointer, the CPU presses the Panic Button as soon as the ES register is loaded.

For all you low-level assembler programmers, here's what the R.ds assignment looks like using HiWord and LongInt:

```
push word ptr [MID+2]
push word ptr [MID]
pop  dx
pop  ax
mov  [R.ds],ax
```

Not earth-shatteringly efficient, but at least a segment register isn't involved. Now here's the same thing again with the Seg operator being used:

```
les di, [MID]  ; Oops, that's torn it...
; We never get here:
; we've already GPFd...
mov ax,es
mov [R.ds],ax
```

This code is a lot more efficient, but completely vulnerable to invalid addresses. I mentioned this bug to Borland many moons ago, back in the days of Borland Pascal 7, but unfortunately it never got fixed. In my opinion, the Seg and Ofs operators should never involve a segment register. This little example shows the dangers of passing invalid addresses around inside your program, even when you aren't expecting to reference them from protected mode.

Once the R variable is initialised, it's relatively plain sailing. All that remains is to make the DPMI call through INT $31. We specify the required interrupt number in the BL register and how many words to copy from the protected mode stack in CX. This facility enables you to issue real-mode interrupt calls which expect one or more parameters on the stack. This isn't the case here so we initialise this value to zero. Finally, the ES:DI registers point to the R data structure and we descend into the bowels of the DPMI server itself... On exit from the DPMI server, the carry flag indicates success or failure and this is used to set the function result as appropriate.

The End Of The Road

So there you have it: a couple of simple little routines which will allow you to get and set the serial number of any drive. If you're still with me so far, the good news is that, like the volume label calls I described last month, all the code here will work under Windows 3.1, Windows 95 and Windows NT.

However, it should be stressed that you should only resort to such hackery if you're writing a 16-bit application that you want to execute under all three of these platforms.

If you're writing an all-new 32-bit program for Windows 95 and/or NT, then for goodness sake use the new 32-bit API routines provided for the purpose. You can use GetVolumeInformation to obtain the volume label and serial number information for a volume, and you can use the SetVolumeLabel call to change the label of a volume. What about setting the serial number from a 32-bit application, I hear you cry? Well, there seems to be no API

call to do this. I suspect that Microsoft don't want you to do it.

Incidentally, while on the subject of volume labels, you might be tempted to suppose that now we've got working `GetMid` and `SetMid` routines, we could just use those to obtain and set the volume serial number. Alas, this isn't the case. These routines operate on the `MIDINFO` data structure which is located on the boot record, the first logical sector of an MSDOS volume. While there is a volume label stored here, it's only the one that was specified when the volume was first formatted. If you subsequently use the DOS `LABEL` command, the appropriate directory entry will be updated but not the boot record information. Consequently, for volume labels we need to stick with the FCB-related nonsense that I showed you last month, while for serial numbers, we use the `MIDINFO` mechanism. Simple, eh? Like I said, it's an excellent example of how not to design an operating system...

Finally, Figure 1 shows our little testbed program running. I took the existing testbed program (see last month) added some code to display volume serial numbers and reorganised it around the `TabSet` control so as to simplify the window layout. Rather than listing the entire program again (most of which is the same) Listing 2 shows the only really important bit: the `OnChange` handler for the drive list control which appears on the `Serial Numbers` page of the notebook. Imaginative use of the `Format` routine allows us to display the selected volume's serial number in the same hi-lo format that's used on the DOS command line.

CD-ROM Detection

Just to round off with, I've added a couple of CD-ROM related routines to the DOSINFO unit. They are both shown in Listing 3. The first of these, `GetCDDriveLetter`, calls the Microsoft `MSCDEX` DOS extensions to determine if a CD-ROM drive is installed. If it is, then the drive letter of the CD is returned as the function result. A zero character (`#0`) is returned if no CD-ROM drive can be found. The routine contains a built-in check to see if `MSCDEX` itself is installed, so it can be called under any circumstances.

The other routine, called `RunningFromCD`, will tell you whether or not the current application is running from the CD-ROM drive. This can be useful if, for example, you're writing an installation program and you want to take special action if the program is being executed from a CD. Note that this routine will always 'fail safe'. In other words, if no CD-ROM drive is present then the `GetCDDriveLetter` routine will always return a zero character. Since this can't possibly correspond to a valid drive letter, the routine will always return `False` in these circumstances.

Next Month

That's all for now. Next month, we'll be delving into the mysteries of floppy disk formatting.

This month's disk contains the source code for the SERNUM.PAS routines (in case you want to keep the serial number stuff separate) together with updated source code for DOSINFO.PAS, which contains all the code we've discussed so far in these last two months.

Dave Jewell is a strange person who enjoys delving around inside the guts of Delphi, Windows and DOS. He works as a freelance consultant, technical journalist and author. His ambition is to get through life without ever writing a database program. You can contact Dave on the internet as djewell@cix.compulink.co.uk or on CompuServe as 102354,1572

➤ *Listing 2*

```
procedure TForm1.DriveList2Change(Sender: TObject);
var
  s: String;
  snum: LongInt;
begin
  s := Copy(DriveList2.Items [DriveList2.ItemIndex], 7, 1);
  snum := GetSerialNumber(Ord(s[1]) - $40);
  TheSerialNum.Caption := Format('%.4x-%.4x', [HiWord(snum), LoWord(snum)]);
end;
```

➤ *Listing 3*

```
function GetCDDriveLetter: Char; assembler;
{ Return the drive letter of the CD-ROM drive (if any);
  if no CD is present, #0 is returned }
asm
  mov  ax,$150B            { do installation check for MSCDEX }
  mov  bx,$ffff            { preset the BX register           }
  int  $2F                 { see if MSCDEX is installed       }
  inc  bx                  { was BX register still -1?        }
  jz   @@1                 { if so, there ain't no CD-ROM !   }
  xor  bx,bx               { clear BX register                }
  mov  ax,$1500            { request starting drive letter    }
  int  $2F                 { result in CX register            }
  add  cl,$41              { normalise into character 0->$41  }
  mov  bx,cx               { result in BX register            }
@1: mov  ax,bx             { result in AX                     }
  mov  ah,0                { clear the high byte              }
end;

function RunningFromCD: Bool;
{ Purpose: True if this application is running from CD }
var
  fName: array [0..255] of Char;
begin
  GetModuleFileName(hInstance, fName, sizeof(fName));
  RunningFromCD := (fName [0] = GetCDDriveLetter);
end;
```