Delphi Internals: How Not To Write An Operating System

Floppy drive fingerprinting and volume labels

by Dave Jewell

Good though Delphi is, there are certain areas where the library support provided by Borland is lacking. Big though Windows might be, there are some useful routines which you just won't find in the Windows API. In this article I'll start plugging some of these holes, with the aim of gradually building up a useful set of reusable routines which you can easily incorporate into your own applications.

Floppy Disk Fingerprinting

As an example of this sort of problem, imagine that you're writing a disk copying program. If the user has got two floppy disk drives, then they can just put the source and destination floppies into the drives, hit the `Enter` button and it all happens. But this scenario assumes that the two floppy disk drives are the same type. On my own machine, I have a 1.44Mb unit set up as drive A: and a 1.2Mb unit set up as drive B:. This is so that I can access old-style 5.25 inch floppy disks when the need arises. Obviously, in that sort of case, you can't do a straight disk copy from one drive to another unless you can determine that both drives are of the same type.

Nowadays, many PCs only have one floppy disk drive. In these circumstances, DOS sets a special internal flag called the "phantom drive flag". It basically fools itself into thinking that it's got two logical drives, A: and B:, whereas only one physical drive exists. DOS keeps track of which logical drive was last accessed and (when higher level software changes which drive it's using) the user gets prompted to insert the required disk corresponding to the new logical drive.

The problem with this approach is that it complicates things for the higher level software. As the potential writer of a floppy-disk copying program, you really do need to know how many floppy drives are out there. If you use the Delphi `DiskSize` routine (in the SYSUTILS unit) to get the size of drive B:, you will only ever get the same disk size as drive A: if you're running on a one-drive system. It won't tell you that there isn't actually a *physical* drive B:.

What's needed is some way of bypassing DOS's logical view of the floppy drives to see what's really going on. Enter Dave's DOSInfo unit. Although I've called it DOSInfo it's really a collection of system-level routines which relate to DOS and disk drives. By eliminating any VCL calls from the code and not using any new Delphi language extensions, you should find that the code is usable from both Delphi and Borland Pascal applications. As we return to this general area in future columns I'll add more functionality to the unit.

There are two routines which relate to floppy disks. The first, `GetFloppyDriveCount`, returns the number of physical floppy drives attached to your PC. It works by retrieving the low-level BIOS equipment word. The flag in bit 0 of the equipment word is set to False if there are no floppy drives installed. If it's set to True, there's at least one floppy drive. In this case, bits 6 and 7 of the equipment word specify the number of installed floppies less one. Why do things like this? Because the original IBM floppy disk controllers supported up to four floppy drives and the number 4 can't be encoded into two bits. It would have been a lot more sensible if IBM had just used the three floppy-related bits as a straight count, but IBM BIOS design has always been regarded as somewhat idiosyncratic...

0	Drive not installed
1	360Kb, 5.25 inch drive
2	1.2Mb, 5.25 inch drive
3	720Kb, 3.5 inch drive
4	1.44Mb, 3.5 inch drive
5	2.88Mb, 3.5 inch drive
Other values are reserved for future use	

➤ *Table 1: Floppy drive type values from CMOS byte at location $10 – the two nibbles in the byte give the types for the first and second floppy drives*

The other routine, which is called `GetFloppyDriveType`, takes a zero-based integer (0 for the first floppy, 1 for the second) and returns the type of the floppy drive as a simple integer indicating the maximum capacity floppy size in Kb. It uses the byte at location $10 in CMOS RAM. This byte is split into two 4-bit nibbles which give the types of the first and second floppy drives. If you have one of those bizarre IBM floppy controllers and more than two drives, then tough! I don't honestly know where the information for those extra drives is stored, or whether it's stored at all. Table 1 shows the possible nibble values are for each drive.

Both the `GetFloppyDriveCount` and `GetFloppyDriveType` routines work perfectly under both Windows 95 and NT.

Volume Labels: Abandon Hope All Ye Who Enter Here...

Anyone who has been programming for MS-DOS since version 1.0

will know all about FCBs. The earliest version of MS-DOS (which wasn't actually developed by Microsoft at all, but was bought in from another company) owed much to CP/M in its design. Opening or closing a file involved setting up a complicated and arcane data structure called a File Control Block (FCB). Things like pathnames didn't exist for the simple reason that sub-directories weren't introduced until version 2.0 of the operating system.

If this all makes you grateful that you've never done any FCB programming, then I've got bad news for you! If you want to manipulate volume labels under Windows, it's very difficult to do so without using FCBs. Read on...

Some time ago, Microsoft implemented a couple of MS-DOS calls whose specific aim in life was to allow an application program to get and set volume labels and volume serial number information. If you look at Listing 1 (which is really a Borland Pascal program), you'll see how things are supposed to work. Basically, these calls use a small data structure to set and retrieve serial number and volume label information. If you were to run this code under DOS, everything would work fine but under Windows it does absolutely nothing. After the `Intr` call, the low bit of the `Flags` register is clear (this corresponds to the processor's `Carry` flag) indicating a successful call to the DOS kernel, but the data structure hasn't been filled in. What's going wrong?

To answer this question, you need to realise that the DOS kernel, because it's written for real mode, can't access data that's higher than the first Mb of RAM. DOS simply can't "see" the global variable whose address we're passing in the `INT $21` call. Windows has a certain amount of DOS functionality built into itself and, if possible, it will field DOS requests directly rather than passing them to the real mode kernel. Unfortunately, the `GetMID` and `SetMID` calls don't seem to be supported by the protected mode part of Windows and consequently the real mode kernel gets invoked.

Now it is possible to get around this problem, but not without a lot of grief. You could, for example, allocate the `MIDINFO` buffer in the first Mb of memory by using `GlobalDOSAlloc`. You'd then have to somehow pass the real mode segment address of this buffer in the `Regs.DS` part of the `Intr` call. Again, this is going to cause problems, because as soon as the `Intr` code actually loads the real mode segment address into the `DS` register, you'll get a GPF because the processor will try and interpret it as a protected mode selector – which it isn't! Is there a way out of this quandary? Well, yes, it's possible to go the whole hog and issue a real mode interrupt call by talking to the DPMI server built into Windows. But if you go down this route, you'll end up with code that's seriously non-portable.

As it happens, the DOS emulator code built into Windows 95 will correctly handle the `GetMID` and `SetMID` calls, but the same code won't work under Windows NT. This means that we have to make a special case of Windows 95, the only platform which correctly emulates the `$440D` functionality in protected mode. However, if you're programming in 32-bits, then you've got another option. You can use the new `GetVolumeInformation` API call. This will neatly return both serial number and volume label information in one call. This routine is implemented under NT, Windows95 and Win32s but not, unfortunately for us, plain vanilla Windows 3.1. To set a volume label, you can use the new 32-bit `SetVolumeLabel` call.

Are volume labels starting to sound like something of a pain in the nether regions? Well, they are. In the end, I decided to bite the bullet and come up with a set of routines that would work in 16-bit and 32-bit software and which would run under NT, Windows95 and Windows 3.1. In order to achieve this goal I had to use – yes, you've guessed – FCBs!

At this point in time, FCBs are virtually obsolete. The DOS emulator code in Windows retains support for only a very small number of FCB-related calls and the main reason these few calls are still supported is so that 'in the know' applications can continue to use FCBs in order to get and set volume labels. Obviously, the Windows File Manager is an example of such an application. Under Windows 95, I imagine that Microsoft now use the `GetVolumeInformation` and `SetVolumeLabel` calls consistently, but I wouldn't be surprised if these routines mapped down onto FCB-oriented DOS calls!

Listing 2 shows the source code for the DosInfo unit. So far, there are only four routines exported from the unit. We've already discussed `GetFloppyDriveCount` and

➤ *Listing 1*

```
program Bomb;
uses WinTypes, WinProcs, WinDOS;
type
  MIDINFO =
    record
      InfoLevel: Word;
      SerialNum: LongInt;
      VolLabel: array [0..10] of Char;
      FileSystem: array [0..7] of Char;
    end;
 var
   mid: MIDINFO;
   regs: TRegisters;
 begin
   FillChar (regs, sizeof (regs), 0);
   regs.ax := $440D;              { IOCTL, subcode $D }
   regs.bx := 3;                  { specify drive C:  }
   regs.cx := $0866;              { device category = 8, Get MID }
   regs.dx := Ofs (mid);          { DS:DX points to MID buffer }
   regs.ds := Seg (mid);
   Intr ($21, regs);
   if not (Odd (regs.Flags)) then
     MessageBox (0, mid.VolLabel, mid.FileSystem, mb_ok);
end.
```

GetFloppyDriveType. The others are GetDriveLabel and SetDriveLabel. The standard convention that I've used here is to specify drive A: as 1, drive B: as 2 and so on. So, if you want to retrieve the volume label for drive D:, you'd pass 4 to the GetDriveLabel routine. If the specified volume has no label an empty string is returned.

The GetDriveLabel routine is quite simple – it doesn't even require an FCB! It relies on the fact that the FindFirst routine in the WINDOS unit can accept a volume label attribute. The only wrinkle in this routine is that if a volume label is greater than eight characters long it will be returned in standard DOS "8.3" format. In other words, DEVELOPMENT becomes DEVELOPM.ENT. So, the routine has to strip the period before returning the volume label.

Eagle-eyed readers will have spotted the fact that I'm using the old WINDOS unit. There are good reasons for doing this. Most importantly, it contains the all-important Intr routine which we use here to call the DOS kernel. To use WINDOS in your programs, just ensure that C:\DELPHI\SOURCE\RTL70 is on your library path and include the unit name in your uses clause.

It's odd that Borland didn't include all of the WINDOS unit's functionality in the new Delphi SYSUTILS unit. Part of the reason for their hesitation might be because some of the routines in SYSUTILS conflict with routine names in WINDOS. FindFirst and FindNext are examples of this, as is the TSearchRec type. Whenever you get a name clash in different units, whether it be a procedure, function, variable, type or constant, you can resolve the conflict simply by prefixing the name of the routine or whatever with the name of the required unit. Thus, TSearchRec becomes WinDOS.TSearchRec etc.

Setting a volume label is considerably more complex. Peeking inside the Windows 3.1 File Manager implementation, I discovered three or four places where FCB-style DOS calls were used to manipulate volume labels. After a bit of thought, I managed to get this down to one.

Volume Labels, Long Filenames and Why The Two Don't Mix

The MS-DOS implementation of volume labels is a kludge. DOS simply uses an ordinary directory entry in the root directory of a particular volume and marks it as a volume label using a special attribute.

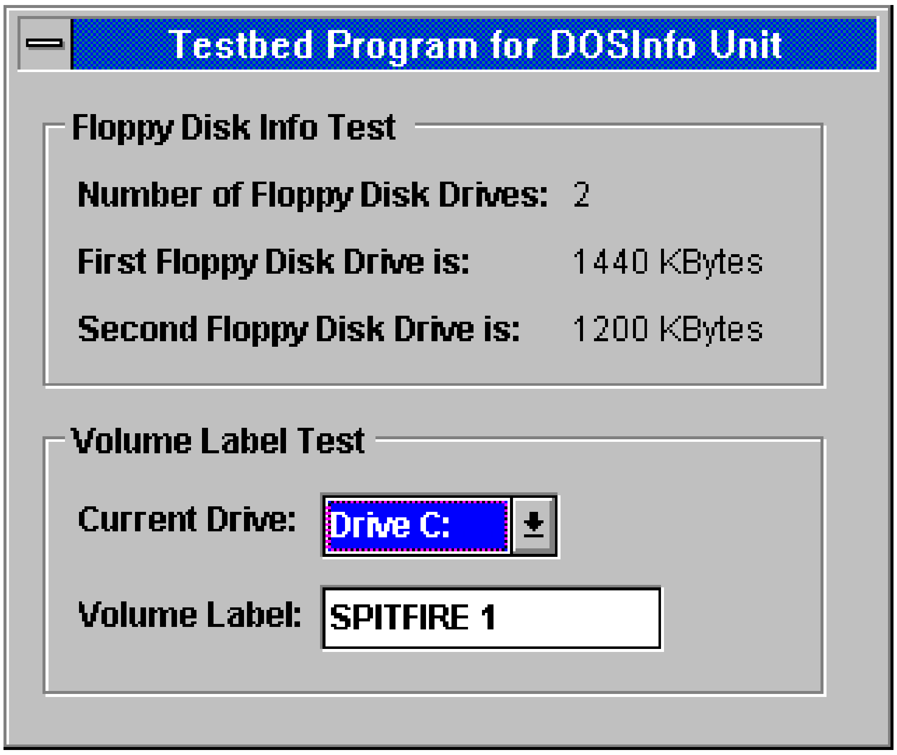
Some third-party software will allow you to create volume labels which contain lower-case characters. Such volume labels cannot be deleted by the DOS LABEL command in the usual manner because of the way that the DOS kernel searches for a directory entry.

In the course of developing the code presented in this article, I discovered that deleting a volume label on drive C: often caused the volume to be reported as having a completely different (and undeletable) volume label. Having spent some time sniffing around with a third-party disk editor, I remembered that Windows 95 uses what look like volume label directory entries in order to store its long filename information. In other words, the Windows 95 long filename support is a hack built on top of a kludge.

Windows 95 uses successive directory entries to store an arbitrarily long filename. Each of these following directory entries has the system, hidden and read-only attributes set in addition to the volume label attribute. Any time you find a system, hidden, read-only, volume label which happens to contain lower-case characters, the chances are almost overwhelming that you're dealing with a Windows 95 long filename entry. In such cases, leave it alone! If you start messing about with it, you could end up corrupting one or more long filenames from the viewpoint of Windows 95. The bottom line is that if you run Windows 95 and Windows 3.1 on a dual boot system, it's quite likely that you won't be able to reliably manipulate the volume labels on your hard disk.

In practice that's not going to be too much of a problem. Why? Because the reason why I'm presenting this volume labelling code in the first place is to use it as part of a floppy disk formatting program that we'll be developing with Delphi.

➤ *Here's our small testbed program in action*


Testbed Program for DOSInfo Unit

Floppy Disk Info Test

Number of Floppy Disk Drives: 2

First Floppy Disk Drive is: 1440 KBytes

Second Floppy Disk Drive is: 1200 KBytes

Volume Label Test

Current Drive: Drive C:

Volume Label: SPITFIRE 1

If you look at the SetDriveLabel routine, it first tries to determine whether a volume label already exists for the specified drive. If so, and assuming that the new volume label isn't the same as the existing one, the old volume label is first deleted by the NukeVolumeLabel routine. Next, if a new label is to be set, the CreateVolLabel code is called. If you want to remove an existing volume label without setting a new one, you can just pass an empty string to SetDriveLabel.

The CreateVolLabel code uses DOS function $3C to create a volume label. Though not an FCB-style call, you can use this routine to create volume labels if you specify 8 as the file attribute. Before calling this code, the MassageVolLabel routine is called to knock the volume label into shape. This routine does four jobs:

- ➢ It truncates volume labels to eleven characters (which is the maximum allowable for the 8.3 file system).
- ➢ It checks to see if any invalid characters were specified as part of the volume label.
- ➢ It appends the string X:\ to the start of the label where X is the required drive letter.

➤ *Listing 2*

```
unit DOSInfo;
interface
uses WinTypes, WinProcs, WinDOS, Strings;
function GetFloppyDriveCount: Integer;
function GetFloppyDriveType(index: Integer): Integer;
function GetDriveLabel(drive: Integer): String;
function SetDriveLabel(drive: Integer; VolLabel: String):
  Integer;
implementation
type XFCB = record  { prehistoric extended FCB - yuck }
  extSig: Byte;     { must be $FF for extended flag }
  extRes: array [0..4] of Byte;     { reserved stuff }
  extAttr: Byte;                    { file attribute }
  extDrive: Byte;                   { drive number }
  extFName: array [0..10] of Char;  { filename }
  extJunk: array [0..24] of Byte;   { rest is irrelevant }
end;
{ Read a single byte from CMOS memory }
function ReadCMOSByte (idx: Byte): Word; assembler;
asm
  mov al,idx               { get the wanted index           }
  out 70h,al               { write address into address reg }
  in  al,71h               { read the drive type into AL    }
  mov ah,0                 { clear the high byte            }
end;
{ Count the number of physical (not logical) floppy drives }
function GetFloppyDriveCount: Integer;
var regs: TRegisters;
begin
  { Get equipment bits }
  FillChar (regs, sizeof (regs), 0);
  Intr ($11, regs);
  if (regs.AX and 1) = 0 then GetFloppyDriveCount := 0 else
    GetFloppyDriveCount := ((regs.AX and $C0) shr 6) + 1;
end;
{ Return the type (max KB capacity) of a given floppy drive }
function GetFloppyDriveType (index: Integer): Integer;
var flopFlags: Word;
  function FlagsToKBytes (flags: Word): Integer;
  begin
    case flags of
      0:     FlagsToKBytes := 0;
      1:     FlagsToKBytes := 360;
      2:     FlagsToKBytes := 1200;
      3:     FlagsToKBytes := 720;
      4:     FlagsToKBytes := 1440;
      5:     FlagsToKBytes := 2880;
      else   FlagsToKBytes := -1;
    end
  end;
begin
  flopFlags := ReadCMOSByte ($10);
  case index of
    0: GetFloppyDriveType := FlagsToKBytes (flopFlags shr 4);
    1: GetFloppyDriveType :=
         FlagsToKBytes (flopFlags and 15);
    else GetFloppyDriveType := 0;
  end;
end;
{ Return the drive label of a specified drive }
function GetDriveLabel (drive: Integer): String;
var i: Integer;
    s: String;
    rec: WinDOS.TSearchRec;
    path: array [0..10] of Char;
begin
  s := '';
  lstrcpy (path, 'X:\*.*');
  path [0] := Chr (drive + $40);     { 1=A, 2=B, etc... }
  WinDOS.FindFirst (path, 8, rec);
  if WinDOS.DOSError = 0 then begin
    for i := 0 to 12 do
      if rec.Name [i] = #0 then break
      else if rec.Name [i] <> '.' then s := s + rec.Name [i];
  end;
  GetDriveLabel := s;
end;
{ Initialise 'fcb' for volume label twiddling - bleurgh ! }
procedure InitLabelFCB (drive: Byte; var fcb: XFCB);
begin
  FillChar (fcb, sizeof (fcb), 0);
  with fcb do begin
    extSig := $ff;       { mark FCB as extended }
    extAttr := 8;        { specify VOLUME attribute }
    extDrive := drive;   { set up drive number (1=A, 2=B..) }
    FillChar (extFName, sizeof (extFName), '?');
  end;
end;
{ Trash any existing volume label }
function NukeVolumeLabel (drive: Byte): Integer;
var fcb: XFCB;
    regs: TRegisters;
begin
  FillChar (regs, sizeof (regs), 0);
  InitLabelFCB (drive, fcb);
  regs.ah := $13;
  regs.dx := Ofs (fcb);
  regs.ds := Seg (fcb);
  MSDos (regs);
  NukeVolumeLabel := regs.al;
end;
{ Massages & validates a user-supplied volume label }
function MassageVolumeLabel (VolLabel: String): String;
var i: Integer;
    str: String;
begin
  str := '';
  MassageVolumeLabel := '';
  if Length (VolLabel) > 11 then VolLabel [0] := Chr (11);
  for i := 1 to Length (VolLabel) do begin
    if StrScan ('*?/\|.,;:+=[]()&^<>"', VolLabel [i])
      <> Nil then Exit;
    if Length (str) = 8 then str := str + '.';
    str := str + UpCase (VolLabel [i]);
  end;
  MassageVolumeLabel := 'X:\' + str;
end;
{ create volume label, assumes there's not one already }
function CreateVolLabel (drive: Byte; volName: String):
  Integer;
var i: Integer;
    regs: TRegisters;
    path: array [0..20] of Char;
begin
  CreateVolLabel := -1;
  StrPCopy (path, MassageVolumeLabel (volName));
  if path [0] = #0 then Exit;        { label was invalid }
  path [0] := Chr (drive + $40);     { 1=A, 2=B, etc... }
  FillChar (regs, sizeof (regs), 0); { safe p-mode programming }
  regs.ah := $3C;                    { specify create file }
regs.cx := 8;                  { set volume label attribute }
  regs.dx := Ofs (path);       { set up pointer to name }
  regs.ds := Seg (path);       { DS:DS is the pointer pair  }
  MSDos (regs);                { do the business...       }
  if not (Odd (regs.Flags)) then begin  { if no carry is ok }
    _lclose (regs.ax);
    CreateVolLabel := 0;
  end;
end;
{ Higher-level volume settings code }
function SetDriveLabel (drive: Integer; VolLabel: String):
Integer;
var err: Integer;
    OldLabel: String;
begin
  err := 0;
  OldLabel := GetDriveLabel (drive);
  { If old and new labels are the same, nothing to do }
  if OldLabel <> VolLabel then begin
    { If got an old label, then delete it }
    if OldLabel <> '' then err := NukeVolumeLabel (drive);
    { If we've got a new label, then set it up }
    if (err = 0) and (VolLabel <> '') then err :=
      CreateVolLabel (drive, volLabel);
  end;
 SetDriveLabel := err;
end;
end.
```

➤ *Listing 3: The unit for the demonstration program shown running in Figure 1*

```
unit Testform;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages,
  Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, DOSInfo, OvcBase,
  OvcEF, OvcSF, OvcTCSim;
type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    FlopTypeB: TLabel;
    FlopTypeA: TLabel;
    FlopCount: TLabel;
    GroupBox2: TGroupBox;
    DriveList: TComboBox;
    Label4: TLabel;
    TheLabel: TEdit;
    Label5: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure TheLabelKeyPress(Sender: TObject; var Key: Char);
    procedure DriveListChange(Sender: TObject);
    procedure FormKeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Char;
begin
  { Set up the various labels }
  FlopCount.Caption := IntToStr(GetFloppyDriveCount);
  FlopTypeA.Caption := IntToStr(GetFloppyDriveType (0)) + ' KBytes';
  FlopTypeB.Caption := IntToStr(GetFloppyDriveType (1)) + ' KBytes';
  for i := 'C' to 'Z' do
    if GetDriveType (Ord (i) - Ord ('A')) = Drive_Fixed then
      DriveList.Items.Add (Format ('Drive %s:', [i]));
  DriveList.ItemIndex := 0;
  DriveListChange (Sender);
end;
procedure TForm1.TheLabelKeyPress(Sender: TObject; var Key: Char);
begin
  Key := UpCase (Key);
end;
procedure TForm1.DriveListChange(Sender: TObject);
var
  s: String;
begin
  s := Copy (DriveList.Items [DriveList.ItemIndex], 7, 1);
  TheLabel.Text := GetDriveLabel (Ord (s[1]) - $40);
end;
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
var
  s: String;
  drive: Integer;
begin
  if Ord (Key) = vk_Return then begin
    Key := #0;
    s := Copy (DriveList.Items [DriveList.ItemIndex], 7, 1);
    drive := Ord (s[1]) - $40;
    if (TheLabel.Text = '') and (GetDriveLabel (drive) <> '') then begin
      if MessageDlg(
       Format('Remove drive label for drive %s: ?', [s [1]]),
       mtConfirmation, [mbYes, mbNo], 0) = mrYes then
        SetDriveLabel (drive, '');
    end
      else SetDriveLabel (drive, TheLabel.Text);
  end;
end;
end.
```

➢ Most importantly, it upper-cases the volume label. Why is upper-casing the volume label so important? Read the sidebar *Volume Labels, Long Filenames and Why The Two Don't Mix* – it will make your hair curl...

The `NukeVolumeLabel` code is the only place where we have to mess with FCBs.

At the time I wrote this code, I feared that I might have to use several FCB calls, and that's why the `InitLabelFCB` routine is separated out – so that it can be called from elsewhere. As it turned out, this wasn't necessary so you could roll these two routines into one to slightly simplify the code.

Conclusion

So that's it! Our eventual aim is to build a complete floppy disk formatting and copying program. Next month we'll move on a bit further. One topic I aim to cover in the next article is reading and writing floppy disk serial numbers – which is another can of worms...

Dave Jewell is a freelance technical journalist, computer consultant and author of *Instant Delphi* from Wrox Press. You can reach Dave by email on the internet as djewell@cix.compulink.co.uk or on CompuServe as 102354,1572