The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

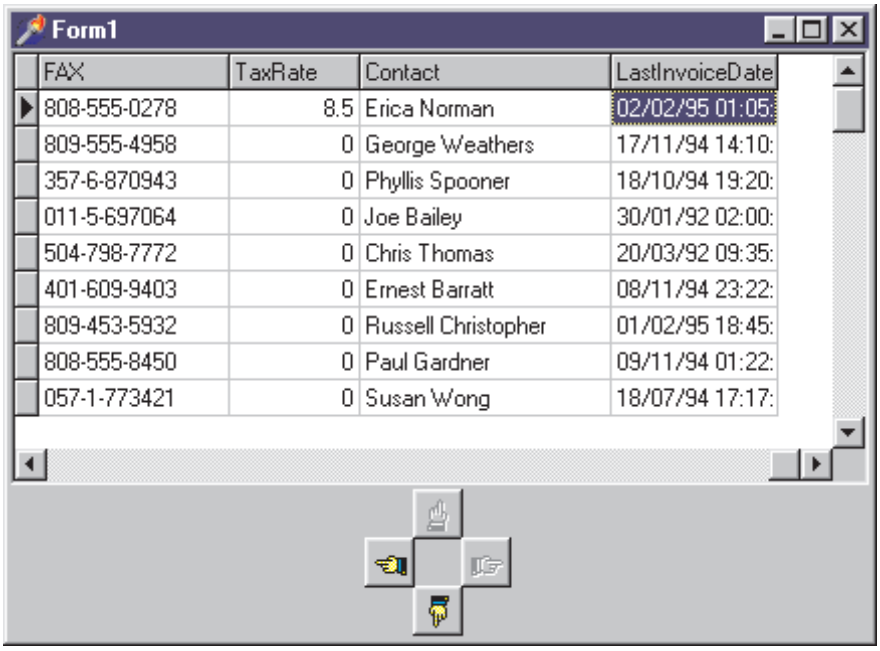
Grid Cell Positioning

Q How can I programmatically change the current cell in a `DBGrid`?

A You can change the currently selected cell in a relative manner with reasonable ease. You use the grid's `SelectedIndex` property to go left and right and the underlying dataset's `Next` and `Prior` methods (as well as `MoveBy`) to go up and down.

GridMov.Dpr on this month's disk shows the idea. It has a quartet of buttons to allow navigating around a grid (see Figure 1). The buttons have `Tag` values to differentiate between them, and there are two shared event handlers to implement what is required (see Listing 1).

The buttons are enabled and disabled as required by two other event handlers: one that is triggered when a field in a different column of the grid is selected and one when the current record changes (see Listing 2).



➤ *Figure 1*

```
procedure TForm1.btnUpAndDownClick(Sender: TObject);
begin
  { Both up and down buttons share this event handler }
  { Move field pointer up or down one record }
  DataSet.MoveBy(TComponent(Sender).Tag);
end;
procedure TForm1.btnLeftAndRightClick(Sender: TObject);
begin
  { Both left and right buttons share this event handler }
  { Move field pointer left or right one field }
  Grid.SelectedIndex := Grid.SelectedIndex + TComponent(Sender).Tag
end;
```

➤ *Listing 1*

```
procedure TForm1.DBGrid1ColEnter(Sender: TObject);
begin
  { When moving from field to field, enable/disable buttons as appropriate }
  btnLeft.Enabled := Grid.SelectedIndex > 0;
  btnRight.Enabled := Grid.SelectedIndex < Grid.FieldCount - 1;
end;
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  { When moving from record to record, enable/disable buttons as appropriate }
  btnUp.Enabled := not TDataSource(Sender).DataSet.BOF;
  btnDown.Enabled := not TDataSource(Sender).DataSet.EOF;
end;
```

➤ *Listing 2*

Application And Window Handles

Q Do you understand how the result of `ShellExecute` can be used? What I would like to do is find the window handle of the application I've just run so I can use window calls like `IsWindowVisible` with it. I don't know the difference between a window handle and application handle.

A First things first. For certain operations on launched applications, you can refer to *Are You Running?* in Issue 28's *Clinic*.

Next, application handles (or, more correctly, task handles in 16-bit Windows and process handles in Win32) are numbers that Windows uses to uniquely identify individual programs. Window handles are numbers that Windows uses to uniquely identify all the windows in existence in the current Windows session. They are different things altogether: remember that one application

```
function EnumFunc(Wnd: HWnd; PWnd: PHandle): Bool; stdcall;
begin
  //Terminate the enumeration
  Result := False;
  //Return the window handle
  PWnd^ := Wnd
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  CString: array[0..Max_Path] of Char;
  SI: TStartupInfo;
  PI: TProcessInformation;
begin
  //Set open dialog to look in Windows directory
  GetWindowsDirectory(CString, SizeOf(CString));
  dlgOpen.InitialDir := CString;
  if dlgOpen.Execute then begin
    //Launch chosen app
    GetStartupInfo(SI);
    Win32Check(CreateProcess(nil, PChar(dlgOpen.FileName),
      nil, nil, False, 0, nil, nil, SI, PI));
    //Wait for main window to initialise
    WaitForInputIdle(PI.hProcess, Infinite);
    //Loop thru all app's windows
    EnumThreadWindows(PI.dwThreadId, @EnumFunc, Longint(@Wnd));
    //Fill listbox with info on the main window
    DescribeWnd(Wnd, lstWindows.Items);
  end
end;
```

➤ *Listing 3*

may contain many different windows.

Finally, you will find that in 32-bit Windows `ShellExecute` is not as handy as the newer `ShellExecuteEx`. `ShellExecute` returns an instance handle, a number that uniquely identifies an application instance within a given address space. In truth, an instance handle is simply the address of the instance's data segment. However, in Win32 land, all application instances are loaded into separate, parallel, address spaces. The upshot of this is that all applications load at the same address in their individual address spaces and so instance handles tend to often have the same values, making them worthless.

`ShellExecuteEx` provides you with a more useful process handle. Process handles are used to identify a process in the Windows session, but unfortunately they are not too useful when trying to find which windows belong to a given process. `CreateProcess` is a more realistically useful API in that it returns both a process handle (and also a process identifier) along with a thread handle (and thread identifier). A thread handle identifies one thread in the current Windows session. `CreateProcess` gives you the handle of the application's primary thread.

The difference between handles and identifiers can be explained like this. Threads and processes are uniquely identified by their identifiers. The handles, however, may not be unique: there may be several process handles which all identify a given process, and the same may be true for threads.

From the thread identifier you can find all the windows created from within that thread using `EnumThreadWindows`. The project Launch.Dpr offers some code to do the required business (see Listing 3). Having launched an application and waited for the first window to start processing messages, it starts looping through the windows created in the thread identified by the thread identifier supplied by `CreateProcess`. The enumeration routine, `EnumFunc`, makes the assumption that the first window found will be the main window and so stores its window handle in the variable whose address was passed through from the call to `EnumThreadWindows`. The enumeration is then promptly stopped by returning `False`.

The next step performed by this demo project is to take the window handle and pass it through to various Windows API calls to find things out about it. This information is written into a listbox thanks to the `DescribeWnd` routine (see Listing 4 and Figure 2). As well as simple analysis, the code makes its own application icon mimic the target window's icon and also flashes the target window using a timer.

➤ *Listing 4*

```
procedure DescribeWnd(Wnd: HWnd; List: TStrings);
const
  FlagStr: array[Boolean] of String = ('not ', '');
var
  CString: array[0..Max_Path] of Char;
  PID, TID: DWord;
  Rect: TRect;
begin
  with List do begin
    BeginUpdate;
    try
      //Check the window is valid
      if not IsWindow(Wnd) then begin
        Add('Window is not valid');
        Abort
      end;
      Add(Format('Window handle = $%x', [Wnd]));
      //Get window caption
      Win32Check(Bool(GetWindowText(Wnd, CString,
        SizeOf(CString))));
      Add(Format('Window caption is "%s"', [CString]));
      //get window dimensions
      Win32Check(GetWindowRect(Wnd, Rect));
      with Rect do
        Add(Format('Window co-ordinates are '+
          '(%d,%d)-(%d,%d)', [Left, Top, Right, Bottom]));
      Add(Format('Window is from app with instance handle'+
        ' of $%x', [GetWindowLong(Wnd, gwl_HInstance)]));
      //Get thread id & process id of owning app
      TID := GetWindowThreadProcessID(Wnd, @PID);
      Add(Format('Window is from app with process '+
        'identifier of $%x', [PID]));
      Add(Format('Window is from app with primary '+
        'thread identifier of $%x', [TID]));
      Add(Format('Window has ID of $%x',
        [GetWindowLong(Wnd, gwl_ID)]));
      Add(Format('The window is %svisible',
        [FlagStr[IsWindowVisible(Wnd)]]));
      Add(Format('The window is %sa Unicode window',
        [FlagStr[IsWindowUnicode(Wnd)]]));
      Add(Format('The window is %senabled',
        [FlagStr[IsWindowEnabled(Wnd)]]));
      Application.Icon.Handle :=
        GetClassLong(Wnd, gcl_HIcon);
      Add('The window''s icon has been set as this '+
        ' application''s icon');
      Add('The window''s caption bar should be flashing')
    finally
      EndUpdate
    end
  end
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  //Flash the window's caption bar
  if IsWindow(Wnd) then
    FlashWindow(Wnd, True)
end;
```

If you are still using Delphi 1 under Windows 3.1x then an alternate project Launch1.Dpr might be helpful. It tries to do as much of the same as possible, using Windows API calls and the `ToolHelp` library.

DLL Failure

Q I have a DLL which was written in Delphi 1 and which has since been recompiled under Delphi 3. Whenever I run an application written and compiled in Delphi 3 and which requires the DLL on Windows NT I get the following error message:

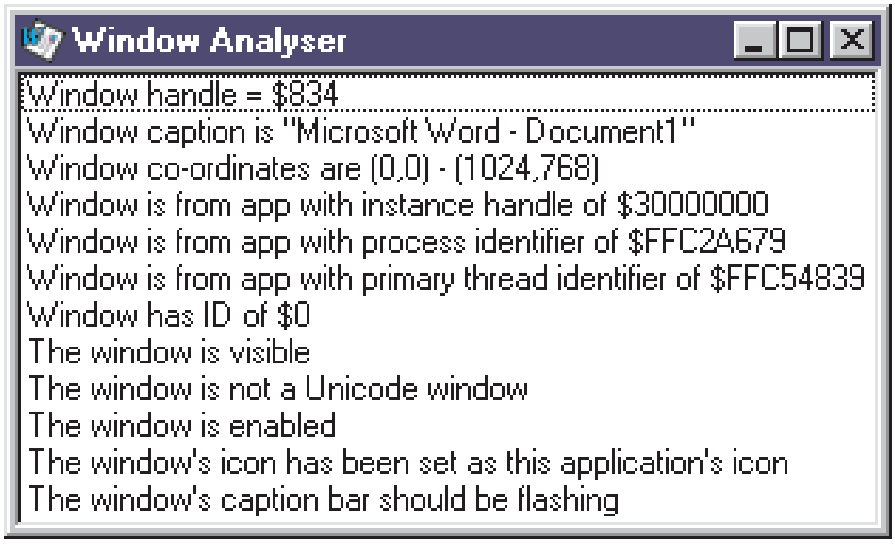
'The dynamic link library XXXX could not be found in the specified path C:\dlb\exe;.;C:\WINNT\System32; C:\WINNT\System...'

I have copied my 'XXXX.DLL' into every directory mentioned in the `path` statement but the error persists. Everything works fine under Windows 95 with just one copy of the DLL in C:\dlb\exe which is my .EXE and .DLL location. What is the problem? Can you help?

A This one is quite a common mistake when moving from 16-bit Windows to 32-bit Windows. Your import declaration *must* have the extension specified in the DLL name. 16-bit Windows insists on no extensions. Windows 95 doesn't mind either way. Windows NT, however, *insists* on an extension, so it's best to use one when writing for 32-bit Windows platforms.

Listing 5 shows source code for a simple DLL. It is written using just a project file, with no Pascal units, for brevity. Listing 6 then shows an import unit containing an appropriate import declaration that some application, or indeed another DLL, can use to link to the routine in Listing 5. You can see how conditional compilation can ensure that the source code will compile in 16- and 32-bit compilers.

To show the DLL working, a project Exe.Dpr is supplied that makes use of the DLL's `Pow` routine. Don't forget to load and compile Dll.Dpr (but don't run it) before trying to run Exe.Dpr.



➤ *Figure 2*

```
library Dll;
uses
  WinTypes;
function Pow(X, Y: Double): Double; {$ifndef Win32}export{$else}stdcall{$endif};
begin
  if X = 0 then
    if Y = 0 then
      Result := 1
    else
      Result := 0
  else
    Result := Exp(Ln(X) * Y);
end;
exports
  Pow index 1;
begin
end.
```

➤ *Listing 5*

```
unit ImportU;
interface
function Power(X, Y: Double): Double;
implementation
{ 16-bit OS require _no_ extension in import declaration }
{ 32-bit: NT requires extension, Win95 doesn't mind }
const
  DLLName = 'DLL'{$ifdef Win32}+'.DLL'{$endif};
{ 16-bit apps typically link by number }
{ 32-bit apps typically link by name }
function Power(X, Y: Double): Double;
  external DLLName {$ifdef Win32}name 'Pow'{$else}index 1{$endif};
end.
```

➤ *Listing 6*

One final point: a 16-bit DLL can only be accessed by another 16-bit module (EXE or DLL). Similarly, a 32-bit DLL can only be accessed by a 32-bit module. In other words, the two projects will need to be compiled by the same Delphi version in order to work.

Playing Videos

Q The `TMediaPlayer` component can play a video file by way of its `FileName` property, as can the 32-bit `TAnimate` component. Of course, as their names suggest, these properties rely on the video file being a stand-alone separate file. I would prefer to merge my videos into my EXE, maybe as resources. How do I do this? And is it much the same for sound files?

A Before looking at the question itself it may be worth going through the steps required to store a resource in an EXE or DLL file, since although they are fairly

```
Cool AVI "c:\delphi3.0\Demos\CoolStuf\Cool.AVI"
Ding WAVE "c:\windows\media\ding.wav"
```

➤ *Listing 7*

```
const ResName = 'Cool'
procedure TfrmAVIResource.Button1Click(Sender: TObject);
begin
  Animate1.ResName := ResName;
  Animate1.Active := True;
end;
```

➤ *Listing 8*

simple, they are not necessarily obvious.

Since this question asks about videos and sound files, I will deal with an AVI and a WAV file. The particular multimedia files in this example scenario are fairly common and should be found on any 32-bit Windows installation with Delphi 3 installed on top. I am assuming Delphi 3 in this case, since the TAnimate component was referred to, and Delphi 2 has effectively been obsoleted by Delphi 3.

First of all, we make a new application and save it. The sample project supplied on this month's disk has been saved with names of AVITest.Dpr and AVITestU.Pas.

Next, choose File|New...|Text to make a new text file. This can be saved as a resource script (.RC file). The sample file is VidRes.RC. A textual resource script is what gets compiled into a binary resource file. Note that it is very important to choose a name that is different to the project name (ignoring the extensions).

Insert the pair of lines from Listing 7 to the text file (changing the paths where necessary). Notice the words AVI and WAVE used to identify the resource types. Also note that you can optionally choose numeric identifiers for the individual resources, but I have used the textual identifiers Cool and Ding.

Now we need to compile this resource script. To help accomplish this, Borland supply a command-line resource script compiler called BRCC32.EXE in Delphi's BIN directory. You might want to copy this into some directory that is on the DOS path to avoid typing lengthy directory paths. To facilitate launching this from the Delphi IDE, choose Tools | Configure Tools... | Add... and fill in the edit boxes like this:

```
Title: Resource Compiler
Program: Command.Com
Parameters: /K BRCC32 $EDNAME $SAVE
```

The reason I launched BRCC32 through COMMAND.COM with a /K parameter is so that the DOS session definitely remains when the compiler has finished its work. This means that any errors will still be visible on the screen and do not immediately disappear. The two terms starting with a $ sign are macros that cause the current editor file to be saved and then cause the full name of the editor file to be passed to BRCC32.

Finally you can press OK and then Close to finish adding the menu item onto the Tools menu. Now choose Tools | Resource Compiler. Hopefully you won't get any error messages, but if you do you will see them. If the compilation succeeds, you will get a file with a .RES extension. The reason for the rule about choosing a resource script with a name different to the project file is that Delphi manages a resource file of its own, named after the project file. If you place any resources in a similarly named resource file, they will be overwritten by Delphi.

Having got a resource file (VidRes.Res) we now need to get it bound in to our executable. This is done with a $R compiler directive. All Delphi form units have a $R directive to bind their associated .DFM form file into the target binary file. It usually looks like this:

```
{$R *.DFM}
```

The asterisk expands to the current editor file name. We need to add another directive into our AVITestU.Pas file like this:

```
{$R VidRes.Res}
```

Now the resources will be present in the EXE upon compilation. So the next task is to work out how to access them.

In the case of the TAnimate component, this job is quite straightforward as there are runtime properties designed for the job. Depending upon whether you tagged your resource with a name or an identifying number (recall that we used a name for both of ours) you can set the ResID or ResName property before setting the Active property to True. Do remember that a TAnimate can only deal with silent AVI files, which could potentially be overly restrictive. Listing 8 shows the very short event handler needed for this.

If the resource is in another module, you can use the ResHandle property to specify the module handle or instance handle of the module containing the video. For example, if you compiled the resources into a resource DLL, you could load the DLL with LoadLibrary or maybe LoadLibraryEx, and assign the returned module handle to ResHandle. If you do this, don't forget to unload the library before termination.

In the case of the media player, things appear to be rather trickier. The component does not surface a way of specifying a resource, and I didn't find a lower-level Windows way of getting around the problem. So, the workaround appears to be to write code to store the video resource into a temporary file (which your application should delete upon termination) and use the normal FileName property.

The code in Listing 9 uses the GetTempPath API to find a suitable temporary directory to put the file in. GetTempPath returns a 0 upon failure or non-zero upon success. Bearing this in mind the return value is typecast into a Bool and

```
const
  ResName = 'Cool';
var
  FileName: String;
procedure TfrmAVIResource.Button2Click(Sender: TObject);
var
  Buf: array[0..Max_Path] of Char;
  FS: TFileStream;
  RS: TResourceStream;
begin
  RS := TResourceStream.Create(HInstance, ResName, 'AVI');
  try
    Win32Check(Bool(GetTempPath(SizeOf(Buf), Buf)));
    FileName := StrPas(Buf) + ResName + '.AVI';
    MediaPlayer1.Close;
    FS := TFileStream.Create(FileName, fmCreate);
    try
      FS.CopyFrom(RS, 0)
    finally
      FS.Free
    end;
    MediaPlayer1.FileName := FileName;
    MediaPlayer1.Open;
    MediaPlayer1.Play;
  finally
    RS.Free
  end
end;
procedure TfrmAVIResource.FormDestroy(Sender: TObject);
begin
  DeleteFile(FileName)
end;
```

➤ *Listing 9*

passed to the Delphi 3 routine `Win32Check`. `Bool` is used instead of `Boolean` because `Bool` considers a bit pattern of `0` to be `False` and any other bit pattern to be `True`. `Boolean` on the other hand only considers the least significant bit, and so an arbitrary non-zero bit pattern might be interpreted as `True` or `False`.

The code creates a `TResourceStream` to access the resource. Note the use of the string `AVI` in the constructor to indicate the resource type. A `TFileStream` is then used to copy it to a file.

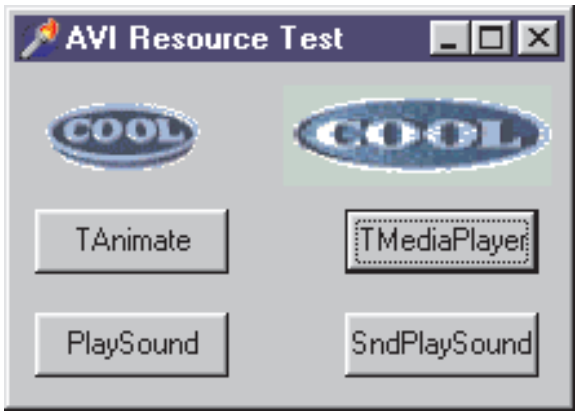
Where the `TAnimate` can only deal with silent AVI files, the media player can deal with any type of file supported by your installed multimedia drivers. So it can potentially deal with AVIs, QuickMovies, RealAudio files, MPEG videos, wave files and so on.

The two approaches to playing the wave resource in this project both involve dedicated multimedia API calls: `SndPlaySound` and `PlaySound`. `SndPlaySound` can play a wave file from disk or from a memory block (via a pointer). `PlaySound` can additionally read directly from a resource. The `TResourceStream` object helps with `SndPlaySound` as, once it has been associated with a resource, you can use the `Memory` property to point to the beginning of the resource data. Listing 10 shows the two pieces of logic. Again, note the use of the string `WAVE` to identify the resource type to `TResourceStream`.

Figure 3 shows the program running the two videos. You can see that the `Transparent` property of the `TAnimate` can help improve the appearance of some AVI clips.

The code that refers to the `TResourceStream` class is restricted to working in 32-bit applications as this class did not exist in Delphi 1. If you wish to get similar logic into a Delphi 1 project, then you will need to use code like Listing 11 to retrieve a pointer to the memory block containing the resource. Listing 11 shows how to play a wave resource in a 16-bit application.

Incidentally, I have also supplied an additional sample project called Video.Dpr that allows you to load arbitrary video files and play them on a resizable form as well as full screen.



➤ *Figure 3*

➤ *Listing 10*

```
procedure TfrmAVIResource.Button3Click(Sender: TObject);
begin
  PlaySound('Ding', HInstance, snd_Resource or snd_Sync);
end;
procedure TfrmAVIResource.Button4Click(Sender: TObject);
begin
  with TResourceStream.Create(HInstance, 'Ding', 'WAVE') do
    try
      SndPlaySound(Memory, snd_Memory or snd_Sync);
    finally
      Free
    end
end;
```

➤ *Listing 11*

```
var
  HResInfo, HGlobal: THandle;
  ResPtr: Pointer;
...
HResInfo := FindResource(HInstance, 'Ding', 'WAVE');
if HResInfo = 0 then Abort;
HGlobal := LoadResource(HInstance, HResInfo);
try
  if HGlobal = 0 then Abort;
  ResPtr := LockResource(HGlobal);
  try
    if not SndPlaySound(ResPtr, snd_Memory or snd_Sync) then
      raise Exception.Create('Failed :-(')
  finally
    UnlockResource(HGlobal)
  end
finally
  FreeResource(HResInfo)
end
```