The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

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

Is This Property Published?

Q Consider the `TCustomPanel` component class, where the `BevelInner` property is protected. Now consider a program that loads many components at runtime, some of which descend from `TCustomPanel`, for example `TPanel1`, `TPanel2`, etc. Some of these `TCustomPanel` descendants have `BevelInner` as a published property, while others do not. This is the question: is there a way to check at runtime if the property `BevelInner` of the arbitrary loaded component is published?

A This sounds like something looking for a generic solution. When a property is published, it has RTTI (runtime type information) generated for it. We can use the support routines in the `TypInfo` unit to check whether RTTI exists for a given property and this will answer the question.

Listing 1 shows a function that takes a class reference and a property name. Its `Boolean` return value suggests whether the property is published in that class. The point about taking a class reference (rather than an object reference) is that you don't need an instance of the class to call this routine. If you want, you can pass the name of a class through to it, as in:

```
if IsPropPublished(TPanel,
  'BevelInner') then…
```

Or, as Listing 1 shows, you can get a class reference for any object by referring to its `ClassType` method.

➤ *Listing 1*

```
function IsPropPublished(AClass: TClass; const PropName: String): Boolean;
begin
  Result := GetPropInfo(AClass.ClassInfo, PropName) <> nil
end;
procedure TForm1.Button1Click(Sender: TObject);
const
  MsgYes    = '%s looks good';
  MsgNoGood = '%s is not based on a TCustomPanel';
  MsgNearly = '%s doesn''t have it published';
var Loop: Integer;
begin
  for Loop := 0 to Pred(ComponentCount) do
    if Components[Loop] is TCustomPanel then
      if IsPropPublished(Components[Loop].ClassType, 'BevelInner') then
        ShowMessage(Format(MsgYes, [Components[Loop].Name]))
      else
        ShowMessage(Format(MsgNearly, [Components[Loop].Name]))
    else
      ShowMessage(Format(MsgNoGood, [Components[Loop].Name]))
end;
```

SQL Cursor Banished

Q I know the SQL hourglass cursor is supposed to portray more than the normal hourglass cursor, but I don't like it. Is it possible to tell the database support code not to use it any more?

A One solution would be to set the `Session`'s `SQLCursor` property to `False` before it is opened. However, according to a message I saw from Benjamin Petersen on the CIX conference system, you have another option. The following statement will make any reference to the offending cursor produce a normal hourglass instead; put the statement in your project source file (include the `Controls` unit in `uses`) or in your main form's `OnCreate` event handler:

```
Screen.Cursors[crSQLWait] :=
  Screen.Cursors[crHourGlass];
```

Bracket Matching

Q Does Delphi offer any bracket matching facility?

A You have several keystrokes you can choose from. In the editor, place the cursor before any type of bracket (normal, square or curly) you are interested in finding the match for. If you are using the default, classic or Brief keystroke mappings, you can use `Ctrl+Q+[`, `Ctrl+Q+Ctrl+[`, `Ctrl+Q+]` or `Ctrl+Q+Ctrl+]`. Default keystroke mapping users can also choose to use `Alt+[` or `Alt+]`. Epsilon mapping users can use `Ctrl+Alt+B`, `Esc+Ctrl+B`, `Alt+)`, `Esc+)`, `Alt+Shift+0`, `Ctrl+Alt+F` or `Esc+Ctrl+F`.

Win32 System Modal Window

Q In Issue 16 you described how to make a system modal window in a 16-bit Windows application. Is there a way to do this in 32-bit Windows also?

A Well, you caught me out there. I did indeed omit any references to Win32 in that answer from a couple of years ago. In Windows 95/98, this is actually quite possible, but NT rather denies us the pleasure as far as I can find.

For non-NT 32-bit systems, we should consider a screen saver. When it starts up, it takes over the desktop area. If the user tries to do anything, one of two things will happen. Either the screen saver will immediately terminate, or a password dialog will appear. If the latter pops up, the user is obliged to enter the correct password before continuing. `Ctrl+Esc` (Start menu), `Ctrl+Alt+Del` (close program) and `Alt+Tab` (switch to other

program) do nothing as they have been disabled.

The screen saver password dialog is not so much a modal window, as a window that is displayed once those system keystrokes have been disabled. The fact that the screen saver occupies the whole screen helps out here: if it only covered a portion of the desktop, you would still be able to switch quite happily to another program by simply clicking one of its windows.

So a Win32 modal window can be simulated by having a maximised window, hiding the taskbar if necessary, and then fooling Windows into thinking a screen saver is running (thereby disabling those keystrokes). The key to making Windows believe it has an active screen saver is to call `SystemParametersInfo`, with a parameter of `spi_ScreenSaverRunning`. This is a constant defined with a value of 97, and even a 16-bit program running on Windows 95/98 can successfully use this call (using the literal value, as the constant is not defined in Delphi 1) to disable the aforementioned system keystrokes.

Listing 2 shows a routine from Modal.Dpr that you can use to start and stop a system 'modal' window by disabling and re-enabling the system keystrokes. Notice the code checks for Windows NT and does not hide the taskbar, since the faked screen saver logic is only effective in Windows 95/98. Checking for Windows NT from a Delphi 1 program was explored in Issue 21, page 56.

An alternative approach that might be more like how we would expect a system modal window to look, would be as follows. Rather than making the 'modal' window a full screen maximised window, we could leave it normal sized by cheating. Just before displaying the window that is to be modal, we can take a snapshot of the Windows desktop, make a new temporary form with a `TImage` on it, and display that maximised. Then, when the 'modal' window comes up, clicking on anything in the background will do nothing (as it will be just part of the image on the form). Listing 3 shows a method of a form in Modal2.Dpr that can be called to launch the form as a system modal window.

```
procedure SetKeyboardAndTaskbarSwitching(Enable: Boolean);
var OldVal: Bool;
const
  TaskBarWnd: HWnd = 0; { Task bar window handle }
  ShowFlags: array[Boolean] of Integer = (sw_Hide, sw_ShowNoActivate);
 {$ifdef Windows}
  spi_ScreenSaverRunning = 97;
 {$endif}
begin
  SystemParametersInfo(spi_ScreenSaverRunning, Word(not Enable), @OldVal, 0);
 {$ifdef Win32}
  if Win32Platform = VER_PLATFORM_WIN32_WINDOWS then begin
 {$else}
  if GetWinFlags and $4000 = 0 then begin
 {$endif}
    if TaskBarWnd = 0 then
      TaskBarWnd := FindWindow('Shell_TrayWnd', nil); { Find task bar }
    ShowWindow(TaskBarWnd, ShowFlags[Enable]); { Hide/show task bar }
  end
end;
```

➤ *Listing 2*

Bad Editor Line Numbers

Q When compiling one project of mine, errors are reported wrongly. The compiler always tells me the problem is several lines earlier than it actually occurs. Obviously, this proves rather irritating. What's going on?

A Figure 1 shows an example of this problem. Basically the issue is that your unit file is a text file. Text files have lines separated by carriage returns and line feeds. You may have edited the file in an editor that didn't use the appropriate line end pattern expected by Delphi's error reporting system.

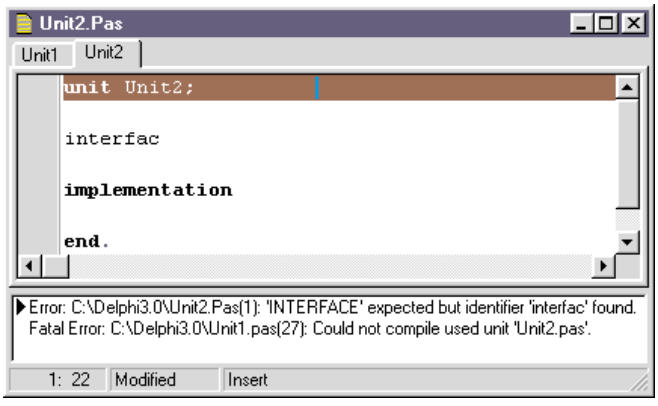
Alternatively you may have pasted a code section in from a badly laid out email message, or some such other source. As a consequence, the compiler doesn't correctly count up the lines to report the error. It thinks that the characters before and after the line terminator are from the same line, assuming the line terminator is just another character.

If you copy all the text in the editor to the clipboard (`Edit | Select All, Edit | Copy`) and then paste it into Notepad you can get a clearer indication of the problem (see Figure 2).

What you can do in Notepad is to select these black squares (printed representations of the decimal characters 10 and/or 13, the line feed and carriage return characters respectively) and press the `Enter` key. That will replace each selected square with the generally accepted pattern for carriage returns and linefeeds. You can then copy the text back into the Delphi editor, and the error reporting should (hopefully) be able to count the lines correctly.

Backspace And New Line

Q Myself and a colleague both have Delphi 3. I have noticed a small but irritating difference between the operation of our



➤ *Left: Figure 1*

➤ *Right: Figure 2*

```
function TFrmModalForm.ShowSystemModal: Integer;
var
  Desktop: TForm;
  DesktopDC: HDC;
begin
  Desktop := TForm.CreateNew(nil);
  try
    { Clear form seems to make less flicker }
    Desktop.Brush.Style := bsClear;
    Desktop.WindowState := wsMaximized;
    Desktop.BorderStyle := bsNone;
    DesktopDC := GetWindowDC(GetDesktopWindow);
    try
      with TImage.Create(Desktop) do begin
        Align := alClient;
        Picture.Bitmap.Height := Screen.Height;
        Picture.Bitmap.Width := Screen.Width;
        BitBlt(Canvas.Handle, 0, 0, Screen.Width,
          Screen.Height, DesktopDC, 0, 0, srcCopy);
        Parent := Desktop;
      end
    finally
      ReleaseDC(GetDesktopWindow, DesktopDC)
    end;
    Desktop.Show;
    { Ensure when anyone clicks on what looks like }
    { another window, all they get is a beep }
    Desktop.Enabled := False;
    SetKeyboardAndTaskbarSwitching(False);
    Result := ShowModal;
    SetKeyboardAndTaskbarSwitching(True)
  finally
    Desktop.Free
  end;
end;
```

➤ *Listing 3*

Delphi installations. When at the beginning of a line in the editor, pressing backspace will take my colleague to the end of the previous line, deleting the carriage return and line feed pair that made the line in the first place. However when I am at the beginning of a line, pressing backspace does nothing. What is wrong on my machine?

A There is nothing really wrong as such. You have probably got the editor set up to use the Brief emulation keystroke mapping. Since Brief did not support backspace at the beginning of a line, then neither does Delphi when it is emulating Brief. If you really need that functionality, I recommend you choose Default or Classic modes.

Incidentally, to be accurate I should really write Brief as BRIEF, since the original DOS editor's name was a somewhat contrived acronym for **B**asic **R**econfigurable **I**nteractive **E**diting **F**acility. Another side note, just for those who are unaware: Brief was originally marketed by UnderWare.

Edit-Less TDBGrid

Q If I make a dataset read-only, then a `DBGrid` connected to it will not allow me to edit any field values, but it sort of looks like it will. What I mean is that when I select a cell, I can press `F2` and get the cursor flashing waiting for me to type, but then the grid does not accept any characters. How do I get the grid to look convincingly uneditable?

A You need to set the relevant option to inform the grid not to manufacture and display its

```
procedure TForm1.FormCreate(Sender: TObject);
{$ifdef Windows} const MAX_PATH = 255; {$endif}
var
  SysDirC: array[0..MAX_PATH] of Char;
  SysDir: String;
  SearchRec: TSearchRec;
begin
  GetSystemDirectory(SysDirC, SizeOf(SysDirC));
  SysDir := StrPas(SysDirC);
  if FindFirst(SysDir + '\*.cpl',
    faAnyFile, SearchRec) = 0 then
    try
      repeat
        ListBox1.Items.Add(UpperCase(SearchRec.Name));
      until FindNext(SearchRec) <> 0;
    finally
      FindClose(SearchRec);
    end;
end;
procedure TForm1.ListBoxDblClick(Sender: TObject);
var Param: String;
    ParamC: array[0..255] of Char;
begin
  Param := 'shell32.dll,Control_RunDLL ' +
    ListBox1.Items[ListBox1.ItemIndex];
  Caption := Param;
  StrPCopy(ParamC, Param);
  ShellExecute(Handle, nil, 'RunDll32.exe',
    ParamC, nil, sw_ShowNormal)
end;
```

➤ *Listing 4*

in-place editor. In the Object Inspector, expand the DBGrid's Options property, and set dgEditing to False, or in code write something like:

```
DBGrid1.Options :=
  DBGrid1.Options - [dgEditing];
```

The situation is reversed with a TStringGrid. Some people do not realise that a string grid *does* support editing if you enable the goEditing entry of its Options property:

```
StringGrid1.Options :=
  StringGrid1.Options +
  [goEditing];
```

Control Panel Invocation

Q My Win32 application has a need to invoke one of the Control Panel applets, to allow the user to change some settings. I don't know how to do this. Do you?

A Well I didn't, but having looked into the problem, I now do. Most of the Control Panel applications are DLLs with .CPL file extensions, which conform to the Microsoft Control Panel applet requirements (specific exported routines that do specified tasks). The others seem to be special cases that Control Panel deals with in different ways, such as Sounds and Users. I will assume that you are only interested in the .CPL files.

We could read up on the Control Panel extension specification, learn about how to call the exported routines, load the .CPL file, locate the routines and call them appropriately, but there are other routes we can take.

To deal with the problem indirectly, we can ask Windows to launch the Control Panel applet. Rather than get bogged down in the details of how to correctly invoke a Control Panel applet, we can delegate the responsibility to a Windows shell routine. Both the 32-bit Shell32.Dll and the Win32 16-bit equivalent Shell.Dll in Windows 95 and Windows NT implement a routine called Control_RunDLL. This takes one PChar parameter indicating the Control Panel applet file name.

As an additional saving in effort, we can call this Shell routine through a helper application. A Win32 application called RunDll32.Exe, and also a 16-bit application called RunDLL.Exe, are available to launch a DLL routine (with a specific parameter list) for you. These both take a command-line that specifies a target DLL and a routine name, along with a string that is passed as the sole parameter to the specified entry point routine. In fact this is exactly what Windows Explorer does when you double click a .CPL file.

So all we need to do to execute, for example, DESK.CPL (the Display Properties Control Panel applet) is to execute the command line:

```
RunDLL32.Exe shell32.dll,
  Control_RunDLL Desk.CPL
```

Listing 4 shows some code from CPanel.Dpr, an application that allows you to test out the idea. It loops through all the *.CPL files and displays their filenames in a listbox. When you double click a file name, the corresponding Control Panel applet is launched and the command line is displayed on the form's caption bar. Figure 3 shows the program after having launched the Display Properties Control Panel applet.

You could also start with the fact that Control.Exe is the (small) stub program that launches Control Panel. It can take a command line parameter indicating which applet to launch. This can be descriptive (eg Control Desktop) or a .CPL name (eg Control Desk. cpl). Examining my Control.Exe in a file viewer leads me to conclude that these descriptive command line parameters are supported (the part in brackets is the equivalent command line parameter):

```
DESKTOP (desk.cpl)
COLOR (desk.cpl,,2)
DATE/TIME (datetime.cpl)
PORTS (sysdm.cpl,,1)
INTERNATIONAL (intl.cpl)
MOUSE (main.cpl)
KEYBOARD (main.cpl @1)
PRINTERS (main.cpl @2)
FONTS (main.cpl @3)
```

Additionally, CONTROL MAIN.CPL @4 gives Power Management settings, and CONTROL MAIN.CPL @5 gives PC Card Settings.

Thanks go to Euan Garden for help in testing out some of this month's answers.

➤ *Figure 3*