The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Character Manipulation

Q I need to get the ASCII number of a one character string. Can you tell me which function I use in Delphi?

A If it is a string variable, and you are sure the length is at least 1 (having used `Length` to check), use `S[1]` to get the first character as a `Char` value. Then you can pass that character to the `Ord` function, or typecast it into a byte value. For example:

```
var
  S: String;
  B: Byte;
...
if Length(S) > 0 then
  B := Ord(S[1]);
  //B := Byte(S[1]);
```

If you need to translate a byte value to a character, either use a `Char` typecast operation, or use the `Chr` function. Incidentally, Windows uses ANSI characters instead of ASCII, which are used by DOS.

SQL Server Problem

Q I've just upgraded my SQL Server installation from version 6.5 to 7.0 and I now have a problem. I added a new row of data in a table containing a date/time field. When I tried to refresh the view of the table in SQL Explorer I got an error message *Syntax error converting date/time from character string*. I also get the error message when I try to delete the row.

A Since your email address seems to be Portuguese, I will assume you are running this setup in Portugal. This is not a BDE problem. You have more than likely forgotten to configure your SQL Server login to Portuguese. The SQL Server client passes a date to the server as a string which is configured according to the client PC's locale. The server expects to receive dates according to the login's locale, which is *not* picked up from anywhere. You must configure this using the server tools provided.

The reported error message will appear when the three character abbreviated month is not the same in Portuguese as it is in English.

DCOM UI Problem

Q I have a DCOM server running on an NT machine. Because it sometimes changes some registry settings I want it to bring up a *Restart Windows* dialog as described in *The Delphi Clinic* in Issue 40. However, no matter what I try, I cannot get the program to display any kind of user interface, even my own forms. What is going wrong?

A This is nothing to do with Delphi as such, more a DCOM configuration question. On the target server machine, you should run the DCOM Configuration application (DCOMCNFG.EXE) and choose the following options: `Applications`, `Object Properties`, `Identity`, `The interactive user`. By default, DCOM servers are launched by `The launching user`, rather than the interactive user. By choosing the interactive user, any user interface manufactured by the application will be displayed on the user's desktop.

Daylight Savings Changeover

Q I need to get the date of the next clock adjustment for Daylight Saving Changes. How can I do it?

A The Win32 API is `GetTimeZoneInformation`, which returns a `TTimeZoneInformation` record containing (potentially) lots of timezone-related information. This includes the descriptive names of the standard time and daylight saving time, along with the difference in minutes between these times and Co-ordinated Universal Time (UTC).

It also gives information on when the changes from daylight to standard time and from standard time to daylight time will occur. These are both supplied as `TSystemTime` record fields of the `TTimeZoneInformation` record. These `TSystemTime` records might describe the transition date in absolute format (where the `wYear`, `wMonth`, `wDay`, `wHour`, `wMinute`, `wSecond` and `wMilliseconds` fields are valid), which means it will be an exact date and time. However, if the `wYear` field is zero, it is using day-in-month format. This also brings the `wDayOfWeek` field into use and enables the routine to indicate, for example, the second Sunday in April, or the last Friday in November, which would then need to be interpreted individually each year. If the `wMonth` field is zero, no transition information is available.

This makes the interpretation of each year's transition dates quite tricky, but these alternate formats are necessary if the user's locale dictates this approach. Listing 1 has some code from the `OnCreate` event handler of the form in a simple sample project called TimeZone.Dpr. The code extracts a whole bunch of timezone-related details and shows them on a form.

```
procedure TTimeZoneInfoForm.FormCreate(Sender: TObject);
var
  RetVal: DWord;
  TZI: TTimeZoneInformation;
  StdBias, DayBias: Integer;
  StdName, DayName: String;
const
  OrdNums: array[1..5] of String =
    ('1st', '2nd', '3rd', '4th', 'last');
  MinsPerDay = SecsPerDay / 60;
begin
  RetVal := GetTimeZoneInformation(TZI);
  if RetVal = $FFFFFFFF then //API call failed
    RaiseLastWin32Error;
  if TZI.StandardName[0] = #0 then //No name information
    StdName := 'standard time'
  else
    StdName := TZI.StandardName;
  if TZI.DaylightName[0] = #0 then //No name information
    DayName := 'daylight time'
  else
    DayName := TZI.DaylightName;
  case RetVal of
    TIME_ZONE_ID_UNKNOWN:
      lblCurrent.Caption := Format(lblCurrent.Caption,
        ['unknown time frame']);
    TIME_ZONE_ID_STANDARD:
      lblCurrent.Caption := Format(lblCurrent.Caption,
        [StdName]);
    TIME_ZONE_ID_DAYLIGHT:
      lblCurrent.Caption := Format(lblCurrent.Caption,
        [DayName]);
  end;
  lblBias.Caption := Format(lblBias.Caption, [TZI.Bias]);
  StdBias := TZI.Bias + TZI.StandardBias;
  lblStdBias.Caption :=
    Format(lblStdBias.Caption, [StdBias, StdName]);
  DayBias := TZI.Bias + TZI.DaylightBias;
  lblDayBias.Caption :=
    Format(lblDayBias.Caption, [DayBias, DayName]);
  lblDayToStd.Caption :=
    Format(lblDayToStd.Caption, [DayName, StdName]);
  lblStdToDay.Caption :=
    Format(lblStdToDay.Caption, [StdName, DayName]);
  if TZI.StandardDate.wMonth = 0 then begin
    lblDayToStd.Caption :=
      lblDayToStd.Caption + 'an unspecified point';
    lblStdToDay.Caption :=
      lblStdToDay.Caption + 'an unspecified point';
    Exit;
  end;
  if TZI.StandardDate.wYear = 0 then //"Day of month" date
    with TZI.StandardDate do
      lblDayToStd.Caption :=
        Format('%s%s on the %s %s of %s',
        [lblDayToStd.Caption, TimeToStr(EncodeTime(wHour,
        wMinute, wSecond, wMilliseconds) +
        DayBias/MinsPerDay), OrdNums[wDay], LongDayNames[
        wDayOfWeek + 1], LongMonthNames[wMonth + 1]])
  else //Absolute date
    lblDayToStd.Caption := lblDayToStd.Caption +
      DateTimeToStr(SystemTimeToDateTime(TZI.StandardDate) +
      DayBias / MinsPerDay);
  if TZI.DaylightDate.wYear = 0 then //"Day of month" date
    with TZI.DaylightDate do
      lblStdToDay.Caption :=
        Format('%s%s on the %s %s of %s',
        [lblStdToDay.Caption, TimeToStr(EncodeTime(wHour,
        wMinute, wSecond, wMilliseconds) +
        StdBias / MinsPerDay), OrdNums[wDay], LongDayNames[
        wDayOfWeek + 1], LongMonthNames[wMonth + 1]])
  else //Absolute date
    lblStdToDay.Caption := lblStdToDay.Caption +
      DateTimeToStr(SystemTimeToDateTime(TZI.DaylightDate) +
      StdBias / MinsPerDay)
end;
```

➤ *Listing 1*

The program can be seen in Figure 1, giving timezone information from my machine. Figure 2 shows the form at design time: the labels' captions have strings ready to be passed to `Format`.

Terminating Programs

Q How do I terminate an external application? From within my application I have launched the *System WAV Player* to play a wave file. Using the `ShellExecute` command, I get the handle to the wave player (or so I believe). I know the following call, which should terminate an application if one knows the caption of its main window, but since I do not know which application is playing the file, I cannot be sure of the caption:

```
PostMessage(FindWindow(
  Nil, 'window caption'),
  wm_Quit, 0, 0);
```

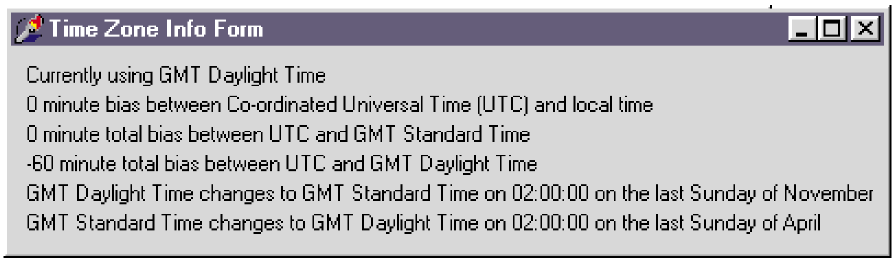
A The `wm_Quit` message normally arrives thanks to an application calling `PostQuitMessage` when the last window is destroyed. You would not normally post it directly to another application.

`TerminateProcess` is an unfriendly way to do the job, but you need to have a process handle. Not a problem if you launched the application with the Win32 APIs `CreateProcess` or `ShellExecuteEx`, as they supply you with the information (unlike the old 16-bit `WinExec` and `ShellExecute`, which in Win32 give you useless values).

Since you are launching the WAV player indirectly, by running the actual WAV file itself, you must use `ShellExecuteEx` (because `CreateProcess` does not understand file associations). This call will give a process handle in the `hProcess` field of its record parameter if you specify see_Mask_NoCloseProcess in the `fMask` field. This process handle can be used to refer to the launched application.

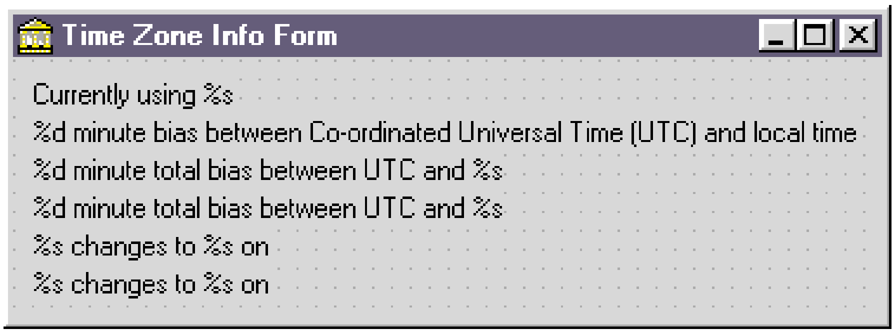
Having got the process handle, you can pass it to a call to `TerminateProcess`. This is a very harsh way to close down an arbitrary process as it gets no opportunity to do any tidying up.

➤ *Figure 1: Extracting time zone information from Windows.*



Currently using GMT Daylight Time
0 minute bias between Co-ordinated Universal Time (UTC) and local time
0 minute total bias between UTC and GMT Standard Time
-60 minute total bias between UTC and GMT Daylight Time
GMT Daylight Time changes to GMT Standard Time on 02:00:00 on the last Sunday of November
GMT Standard Time changes to GMT Daylight Time on 02:00:00 on the last Sunday of April

➤ *Figure 2: The time zone form at design time.*



Currently using %s
%d minute bias between Co-ordinated Universal Time (UTC) and local time
%d minute total bias between UTC and %s
%d minute total bias between UTC and %s
%s changes to %s on
%s changes to %s on

Preferably, you should try and terminate programs in a more friendly, and 'clean' way.

More information on launching applications, and waiting for them to finish, can be found in *The Delphi Clinic* in Issue 16, p52 and Issue 20, p55. Page 54 of Issue 32 discusses the difference between an instance handle, as returned by ShellExecute, and a window handle. It also describes how to get the window handle of a launched application's main window, by using EnumThreadWindows to enumerate through all the windows owned by the main thread of the application just launched. Alternatively, EnumWindows can be used to iterate over all windows in all threads in the system, and GetWindowThreadProcessID can be used to verify if it belongs to the process in question.

Unfortunately, whilst CreateProcess returns a thread handle, thread ID, process handle and process ID, ShellExecuteEx only returns a process handle. The relevant API calls (EnumThreadWindows and GetWindowThreadProcessID) take IDs, not handles. So ShellExecuteEx gives you one advantage (working with file associations) but leaves you with a problem (not being able to identify the launched application's windows).

So, at this point, if you launch the viewer with ShellExecuteEx, the best I can come up with is the unfriendly call to TerminateProcess. This will require the process handle in question to have PROCESS_TERMINATE access under Windows NT, which might mean you will need to call DuplicateHandle to get a new version of the process handle with relevant access rights.

But now, let's say that you work out an appropriate command-line that launches the program with CreateProcess, giving you all the handles and IDs available. How do we do a so-called clean process termination? According to Microsoft's MSDN article Q178893, we should iterate through all the top-level windows of the application and use PostMessage to post a wm_Close message to each one (not wm_Quit). This allows the application to close as if the user closed all the windows. You should then wait for a certain interval, sufficient for the user to deal with any confirmation dialogs that come up. If the process is still around, you can then legitimately use TerminateProcess.

To wait for the a given timeout, you can either call WaitForSingleObject, passing in the process handle and a timeout value, or you could send the messages to the target windows with SendMessageTimeout instead of PostMessage. The Windows Task Manager seems to wait about 10 seconds before offering the chance to brashly terminate the process (or wait yet another 10 seconds).

The sample project on the disk, TermApp.Dpr, shows both scenarios. A button on the form will allow you to launch a program with ShellExecuteEx, and another button will terminate it with TerminateProcess.

Alternatively, some other buttons will use CreateProcess to launch the program and a combination of APIs to terminate it. The first pair of buttons are quite straightforward, but the second pair warrant some looking at. Listing 2 shows the event handlers and Figure 3 gives you an idea of what happens if the application does not terminate within ten seconds. Does this look familiar at all?

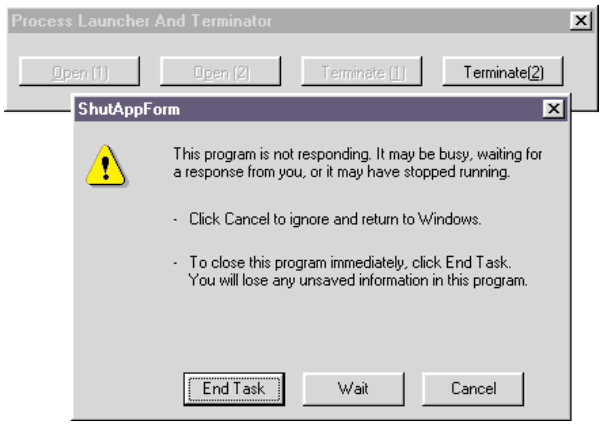
Computer Name In Registry

Q Do you know how to read from

```
HKEY_LOCAL_MACHINE\SYSTEM\
  CurrentControlSet\Control\
  ComputerName\ComputerName\
  ComputerName
```

➤ *Listing 2*

```
procedure TMainForm.btnLaunch2Click(Sender: TObject);
var
  SI: TStartupInfo;
  PI: TProcessInformation;
begin
  if dlgOpen.Execute then begin
    GetStartupInfo(SI);
    Win32Check(CreateProcess(nil, PChar(dlgOpen.FileName),
      nil, nil, False, 0, nil, nil, SI, PI));
    //Save process information
    HProcess := PI.hProcess;
    ProcessID := PI.dwProcessId;
    ThreadID := PI.dwThreadId;
    btnLaunch1.Enabled := False;
    btnLaunch2.Enabled := False;
    btnTerminate1.Enabled := False;
    btnTerminate2.Enabled := True;
    WaitForInputIdle(HProcess, Infinite);
  end
end;
function EnumFunc(Wnd: HWnd; TargetPID: DWord): Bool;
  stdcall;
var PID: DWord;
begin
  GetWindowThreadProcessID(Wnd, @PID);
  if PID = TargetPID then
    PostMessage(Wnd, wm_Close, 0, 0);
  Result := True;
end;
function CheckAppClosed(Process: THandle): Boolean;
var OldTime: TDateTime;
const
  mrEndTask = 100;
  mrWait = 101;
begin
  Result := False;
  OldTime := Now;
  //Loop till either 10 sec is up, or program has terminated
  repeat
    //Do quick check on the app, but not long
    //enough to block (hang) this UI thread
    case WaitForSingleObject(Process, 100) of
      Wait_Object_0: Result := True;
      Wait_Failed: RaiseLastWin32Error;
    end;
    //Stop UI from hanging
    Application.ProcessMessages;
    //If user wants to shut, then fine
    if Application.Terminated then
      Break;
  until Result or (Now > OldTime + 10 / SecsPerDay);
  if not Result then //timeout has passed
    case ShutAppForm.ShowModal of
      mrEndTask :
        begin
          TerminateProcess(Process, 1);
          Result := True
        end;
      mrWait   : {do nothing - we will loop again} ;
      mrCancel : Result := True;
    end
end;
procedure TMainForm.btnTerminate2Click(Sender: TObject);
begin
  EnumWindows(@EnumFunc, LPARAM(ProcessID));
  //May need to do this whole 10 sec wait repeatedly
  repeat until CheckAppClosed(HProcess);
  btnLaunch1.Enabled := True;
  btnLaunch2.Enabled := True;
  btnTerminate1.Enabled := False;
  btnTerminate2.Enabled := False;
end;
```


Process Launcher And Terminator

ShutAppForm

This program is not responding. It may be busy, waiting for a response from you, or it may have stopped running.

- Click Cancel to ignore and return to Windows.

- To close this program immediately, click End Task. You will lose any unsaved information in this program.

➤ *Figure 3: Programmatic process termination.*

in the registry? I've tried, but because the `TRegistry` object uses `HKEY_CURRENT_USER` as its root, the code I have used in the past does not work. My applications need to know what machine they are running on.

A In order to access a key under a root other than `HKEY_CURRENT_USER`, you need to use the `RootKey` property of the `TRegistry` (or `TRegIniFile`) object. So Listing 3 would do it.

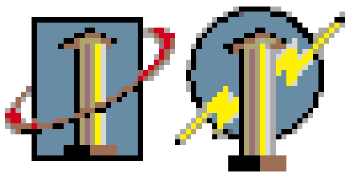
On Windows NT, you must have appropriate access in your program to access the registry.

But after sorting out how to access that key in the registry, I should point out that the approved way of identifying the current computer name is to not use the registry at all. Instead, you should use the relevant Win32 API, `GetComputerName`. Listing 3's `ComputerName` function can now be rewritten as shown in Listing 4.

New Delphi 5 Features

Q I read your review of Delphi 5 on last month's disk and saw coverage of a number of mostly high-level features. Can you give any details of some of the more low-level changes that you know of?

➤ *Figure 4: Delphi 5's new icons.*



A Having seen a pre-release version of Delphi 5 being demonstrated a few times at the Inprise Conference recently, I did learn a few more things that I can share with you. In no particular order, here is a list of those that I can think of at the moment that I saw or heard during talks by R&D members Chuck Jazdzewski and Eddie Churchill.

First of all, the Delphi 5 icon has changed, as has the default icon used for your applications. Figure 4 shows the pair of them, with the Delphi IDE icon to the left.

The Object Inspector has been trained to remember much better which properties were selected, and which properties were expanded, as you switch between components. For example, let's say you are looking at a `TMemo` component. You can expand the `Anchors` property, and the `Font` property, and also the `Style` sub-property of the font. You can select the `Size` sub-property of the font, then select a `TColorDialog` component which has none of these properties. If you then re-select the `TMemo`, the `Anchors`, `Font` and `Style` properties will still be expanded, and `Size` will still be selected.

Whilst mentioning fonts in the context of the Object Inspector, something else springs back to mind. The Object Inspector now allows custom drawn lists of property values (as was discussed in the review), and Delphi will do a reasonable job for cursor, colour, brush style and pen style properties, it does not (by default) give you a WYSIWYG display of font names to choose from. The font name property list looks just the same as it always has done.

The bracketed 'by default' phrase was used because there is built-in support for this, but it is disabled by default. When enabled, all fonts on the system get enumerated, loaded and drawn when the property value list is dropped down. On slower machines, or machines with vast quantities of fonts, this will take some time to do. But if you want to test it out, the job is simple.

The `DsgnIntf` unit now defines a global variable called `FontNamePropertyDisplayFontNames`, which is set to `False`. If you make a small unit that sets this to `True`, either in its `initialisation` section, or in a parameter-less `Register` procedure (which is declared in the interface section), then you can install it into the IDE in a package.

The easiest way is to save the unit, choose `File | Open...` and locate the Delphi User Package DCLUSR50.DPK in Delphi's Lib

```
function ComputerName: String;
begin
  Result := 'Unknown';
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if OpenKey('System\CurrentControlSet\Control\ComputerName\ComputerName',
        False) then
        Result := ReadString('ComputerName')
    finally
      Free
    end;
end;
...
ShowMessage(GetComputerName)
...
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

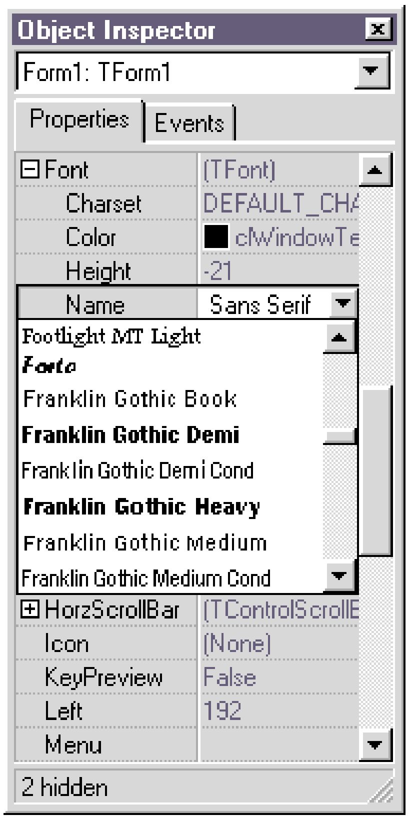
```
function ComputerName: String;
var
  Buf: array[0..MAX_COMPUTERNAME_LENGTH] of Char;
  Len: DWord;
begin
  Len := SizeOf(Buf);
  if GetComputerName(Buf, Len) then
    Result := Buf
  else
    Result := 'Unknown';
end;
```

directory. When the package opens up, click on the contains node in the tree view, press the Add button, locate the unit with the Browse... button and press OK. If the Install button is enabled on the package editor then press it, otherwise press the Compile button. Now find a component with a Font property, expand it and drop down the list of values for the Name sub-property. Figure 5 shows the result.

Where appropriate, components that have an ImageIndex property to pick an image from an image list component, also give visual feedback in the property editor. Figure 6 shows the ImageIndex property of a TToolButton in a TToolBar whose Images property has been connected to a TImageList component populated with bitmaps.

Yet another new feature of the Object Inspector appears to be undocumented. Now that properties are categorised, you can right-click the Object Inspector, choose View and then do one of several things. The popup menu allows you to enable or disable

➤ *Figure 5: The font name property with visual feedback.*



Object Inspector	
Form1: TForm1	
Properties	Events
⊟Font	(TFont)
Charset	DEFAULT_CHA
Color	■ clWindowTe
Height	-21
Name	Sans Serif
⊞HorzScrollBar	(TControlScrollE
Icon	(None)
KeyPreview	False
Left	192
Menu	
2 hidden	

Footlight MT Light
Forte
Franklin Gothic Book
Franklin Gothic Demi
Franklin Gothic Demi Cond
Franklin Gothic Heavy
Franklin Gothic Medium
Franklin Gothic Medium Cond

categories individually, enable all categories, no categories, or to toggle the categories. This last option means disabling the display of all categories currently being displayed and then enabling all those that were hidden.

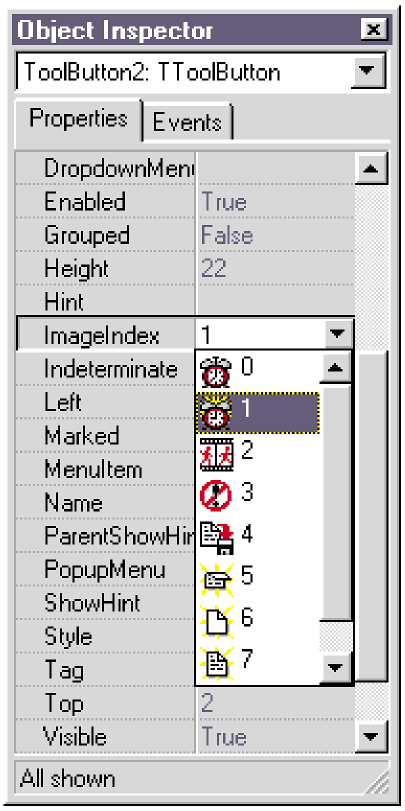
The undocumented facility is holding down the Ctrl key when choosing a category. This causes all categories to be hidden *except* the one you clicked. This saves you going through the menu to disable all categories, and then going through the menu again to enable the one you want to see.

Incidentally, in Dave Jewell's *Second Opinion* section of the Delphi 5 review, he was unhappy about properties being able to be displayed in several categories. Inprise are actually making a plus point out of this. Delphi 5 is the only tool that will do this, and it should ultimately prove handy. Take the Caption property for example. It is textual and so localisable, and it appears in the Locale category. However it is also visual and so appears in the Visual category. Depending upon what you are working on, you might immediately think of either of these categories to check for Caption. Delphi 5 makes things easy because the property lives in both categories.

Finally on the subject of the Object Inspector, it now handles published Int64 properties.

A new item in the File | New... dialog is a Console App Wizard. This is not an interactive wizard, but it makes light work of setting up a GUI-less application. When you invoke it, you get what is shown in Listing 5.

When working with OnKeyDown and OnKeyUp handlers, the Key parameter is a Windows virtual key code. Those in the know are aware that constants for these are defined in the Windows import unit. They may also know how to coax information out of the Win32 API reference help file to describe and name these constants. Certainly, in Delphi 2, 3 and 4, the suggestion topics to look at in the OnKeyDown/OnKeyUp help page have been completely unhelpful in this



Object Inspector	
ToolButton2: TToolButton	
Properties	Events
DropdownMenu	
Enabled	True
Grouped	False
Height	22
Hint	
ImageIndex	1
Indeterminate	
Left	
Marked	
MenuItem	
Name	
ParentShowHint	
PopupMenu	
ShowHint	
Style	
Tag	
Top	2
Visible	True
All shown	

➤ *Figure 6: A meaningful ImageIndex property... at last!*

regard. Pleasantly, the Delphi help file now has a page discussing and listing these codes, and the help for the aforementioned events links to it.

The TSplitter component has AutoSnap and MinSize properties. MinSize dictates the smallest size that panes either side of the splitter can be shrunk to. AutoSnap dictates whether the size of a neighbouring pane will be set to 0 if the user tries to make it smaller than MinSize.

Menu components have had some nice enhancements made to them. They can now use separate image list components for any submenus you choose.

They can also now work out their own hotkeys (the underscored letters, normally set up with an & character). This can be done throughout an entire menu

➤ *Listing 5*

```
program Project2;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
  // Insert user code here
end.
```

structure automatically (before a menu is displayed), or selectively done on individual menus and submenus, using the AutoHotkeys property. Additionally, if you are dynamically building a menu, you can manually kick-start this process with the RethinkHotkeys method. This means that you can design a menu as shown in the menu designer in Figure 7, and at runtime it will look like Figure 8.

Additionally, menus have another property, AutoLineReduction, that works in a similar way (and has a related routine RethinkLines) and ensures that a menu does not start or end with a separator line, or have two of them next to each other. To make dynamic menu creation easier, menu items have methods for adding new separator lines in various places, and some other useful routines you can look for.

Popup menus can now control the popup animation as supported by Windows 98 and Windows 2000 with their MenuAnimation property.

RTTI support has been up-rated by a number of new things in the TypInfo unit. Firstly, there is a new TTypeKind value for unsigned long types, something which should have appeared in Delphi 4, what with Cardinal being defined correctly there, and also the addition of LongWord.

More importantly, however, is the addition of a number of new property reader and writer routines. Some of these are brand new, to cater for reading and writing enum, set and object properties, without having to use GetOrdProp and SetOrdProp in conjunction with a typecast. GetEnumProp, for example, returns a string, which is the textual representation of the enum value. Previously, to get an enumerated property value would require a call to GetOrdProp and then a call to GetEnumName.

Most of the new routines are overloaded versions of the existing routines, to provide easier access to the property values. These easy access routines are designed to take a property name as a string, rather than require you to get a pointer to the TPropInfo property information record. But be warned! None of these easy access routines do error checking. They expect the named property to exist, and will likely cause an Access Violation if this is not so. If you are unsure, or are writing generic code, be sure to call the new IsPublishedProp first.

To help generic property reading and writing, GetPropValue and SetPropValue do the job of most of the other routines, but take or return the property value as a Variant.

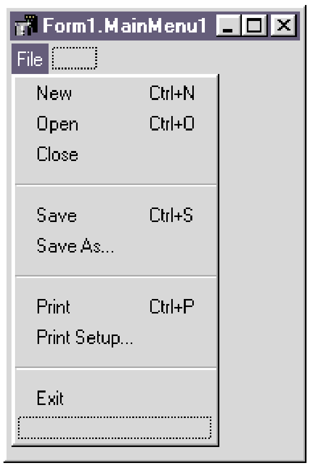
The Dialogs unit has a new global variable called ForceCurrentDirectory, which can be set to True to avoid Windows 98 and Windows 2000 defaulting open and save dialogs to the My Documents folder when the initial directory is blank. Instead, this causes them to stick to the current directory. Because this variable did not exist in Delphi 4's VCL, the Delphi 4 IDE fell foul of this irritation, often prompting you for projects to open from My Documents.

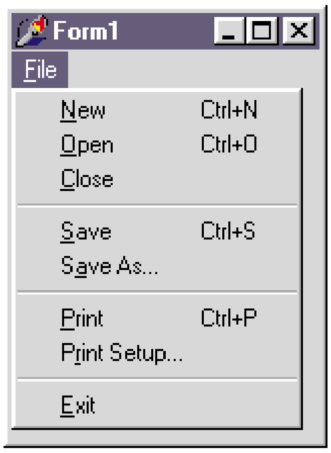
Finally in this list (but certainly not finally in what is new in the product), a word or two about frames. Frames were introduced to improve on the idea of component templates. Component templates are a good convenience measure for reproducing a bunch of components with set properties and event handlers. However, if you want to change the original template, it will have no effect on any copies of the original template which you have made. Frames use a modified version of form inheritance technology to ensure that any change which is made to the original frame is replicated throughout all places where the frame was used.

If you are planning on making several uses of some component that has a heavy impact on the DFM file (for example an image component with a bitmap installed in it), a frame is a good way of easing the byte burden on your EXE. If you put the image in a frame, and use the frame multiple times, the image data will only be stored once in the frame's DFM.

Component writers must follow certain rules to ensure your components will work correctly with frames. Firstly, you must ensure your component supports form inheritance. By default, a component's ComponentStyle has the csInheritable flag in it. TNotebook and TTabbedNotebook remove this flag from the set and so do not support form inheritance, and by implication will not work with frames. If you remove this flag, your component will not work with frames. When a component without the csInheritable flag (such as TNotebook) is dropped on a frame, you are presented with the message: *TNotebook is marked as not supporting form inheritance and frames, and cannot be used in a frame.*

The second point is that when a frame instance has been dropped onto a form designer, no new child controls can be given to the frame by the user (anything dropped on the frame instance will be a child of the underlying form). Any property or component editors must uphold this rule. If you write property or component editors that manufacture new child controls then you are obliged to make a few checks before going ahead.





➤ *Left, Figure 7: No hotkeys in sight.*

➤ *Right, Figure 8: Hotkeys automatically assigned.*

Best of all is to only allow access to the property/component editor if the component is not in a frame instance sitting on a form. To check this, you need to ensure that both the component itself, and also underlying root object (the form, if looking at a form designer, or the frame when looking at a frame designer) can accept child controls. A TPageControl does this, and so the New Page item from its popup menu is not displayed if the component is in a frame on a form.

So, let's say we are writing a hypothetical component called a TNewPanel, which has one component editor menu item whose job is to create a child button in the panel. This component editor item must be disabled if the TNewPanel instance is in a frame on a form (a so-called inlined frame). The component editor could look like Listing 6.

Incidentally, some component developers will often refer to Designer.Form in their property or component editors when they should in fact be referring to Designer.GetRoot. GetRoot returns the underlying form when the component is in a form designer. Similarly, it returns the frame on a frame designer and the data module for a data module designer.

Many developers assume (wrongly) that Designer.Form returns the same thing as Designer.GetRoot, but this is only true when editing forms in a form designer. For data modules, web modules and frames, Designer.Form returns a reference to the placeholder form that the IDE manufactures in order to show you what components are on the data module etc. In the case of a frame, it is an instance of a TWinControlForm, and for a data module, service, or web module it is a TDataModuleDesigner.

```
procedure TNewPanelEditor.ExecuteVerb(Index: Integer);
var Btn: TButton;
begin
  if Index = 0 then begin
    Btn := TButton.Create(Designer.GetRoot);
    Btn.Name := Designer.UniqueName('Button');
    Btn.Caption := TimeToStr(Time);
    Btn.Left := Random(TControl(Component).Width - Btn.Width);
    Btn.Top := Random(TControl(Component).Height - Btn.Height);
    Btn.Parent := TWinControl(Component);
    Designer.Modified
  end
end;
function TNewPanelEditor.GetVerb(Index: Integer): string;
begin
  Result := 'Do it'
end;
function TNewPanelEditor.GetVerbCount: Integer;
begin
  Result := 0;
  //The one component editor verb implemented here depends
  //upon the component not being in an inlined frame instance
  if not IsInInlined then
    Result := 1
end;
```

➤ *Listing 6*

Acknowledgements

Thanks go to Inprise's very own Steve Axtell for help with this month's database problem.