The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, at clinic@blong.com or write/fax us at The Delphi Magazine

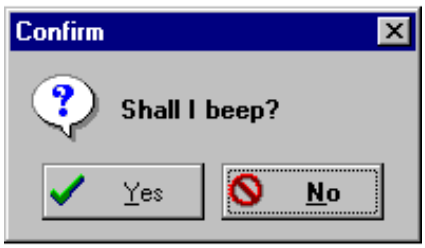
Message Dialog Buttons

Q I want to use `MessageDlg` but it is too inflexible. I can handle the fact that the caption cannot be fully customised, but I need to be able to specify which will be the default button. Is there a trick to this or must I use the `Application` object's `MessageBox` method?

A I can see your point. If you were to bring up a message box saying *Format hard disk?* you wouldn't want the default to be `Yes`. The routines as they stand do not support custom default buttons: they always default to the first one and the order of the buttons is not under your control. However, `MessageDlg`, `MessageDlgPos`, `ShowMessage` and `ShowMessagePos` all make use of the `CreateMessageDialog` routine which is called to make the message form (but not display it). We can re-implement `MessageDlg` and `MessageDlgPos` by cribbing the Borland code and inserting some extra lines to do the desired deed. The new code will just loop through all the controls on the form hunting out the intended default and then make it so.

Listing 1 shows the `MsgDlgs` unit that implements the two updated routines. Notice that they both take an extra parameter of type `TModalResult`: the same sort of values as they return when a button is pushed. So, if you wanted a message box with `Yes` and `No` buttons, with a default of `No`, you could use the code snippet in Listing 2 which would give you the message box in Figure 1 (if you were running Delphi 1). *[See also Steven Colagiovanni's article in Issue 18. Editor]*

➤ *Figure 1*



Win32 Data Type Problems

Q I wish to use a "critical section" in my program (I have encountered these in C++). When I try to declare a variable of type `CRITICAL_SECTION` (as specified by the online help) I get a compiler error. What's wrong?

A The Win32 help file supplied with Delphi 2 and 3 is the same file that Microsoft give Borland to distribute: Borland do not change it. This is why all the API help is given in C syntax. When Borland translate the C Windows import headers into Pascal, they sometimes change the data types to make them more Pascal like. One example of this is `CRITICAL_SECTION` which in Delphi is actually defined as `TRTLCriticalSection`.

The fact that you didn't know this implies you are using Delphi 2, since Delphi 3's `Code Parameters` option would give the game away. When you type in an API that uses a critical section (such as `EnterCriticalSection`) and type the open parenthesis a tooltip appears showing the full parameter list as it appears in its Delphi declaration (see Figure 2).

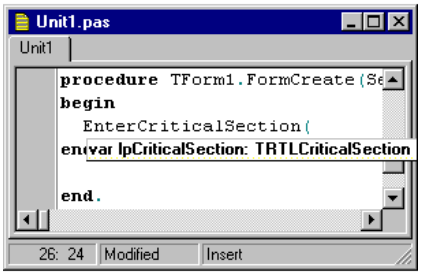
If you are using Delphi 1 or 2 then you will need to look at the available source code. The most common APIs are declared in Delphi 1's `WinProcs` unit and all the

➤ *Listing 1*

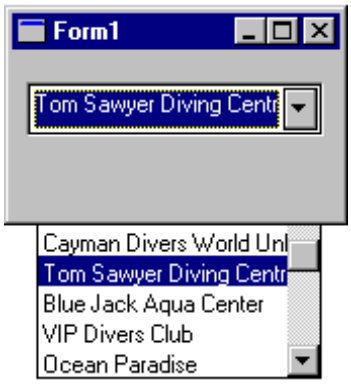
```
function MessageDlgDef(const Msg: string; AType: TMsgDlgType; AButtons:
  TMsgDlgButtons; DefButton: TModalResult; HelpCtx: Longint): TModalResult;
begin
  Result := MessageDlgDefPos(Msg, AType, AButtons, DefButton, HelpCtx, -1, -1);
end;
function MessageDlgDefPos(const Msg: string; AType: TMsgDlgType; AButtons:
  TMsgDlgButtons; DefButton: TModalResult; HelpCtx: Longint; X, Y: Integer):
  TModalResult;
var I: Integer;
begin
  Result := 0;
  with CreateMessageDialog(Msg, AType, AButtons) do
    try
      HelpContext := HelpCtx;
      if X > -1 then Left := X;
      if Y > -1 then Top := Y;
      ScaleBy(Screen.PixelsPerInch, 96);
      { Change the default button }
      for I := 0 to Pred(ComponentCount) do
        if Components[I] is TButton then
          if TButton(Components[I]).ModalResult = DefButton then
            ActiveControl := TButton(Components[I]);
      Result := ShowModal;
    finally
      Free;
    end;
end;
```

➤ *Listing 2*

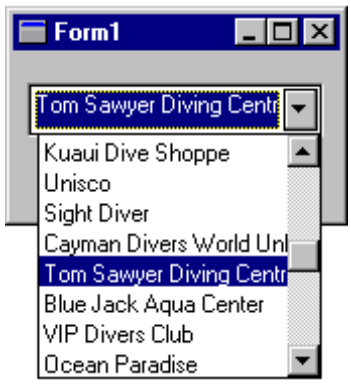
```
if MessageDlgDef('Shall I beep?', mtConfirmation, [mbYes, mbNo], mrNo, 0) = mrYes
then
  MessageBeep(Cardinal(-1))
```



➤ *Figure 2*

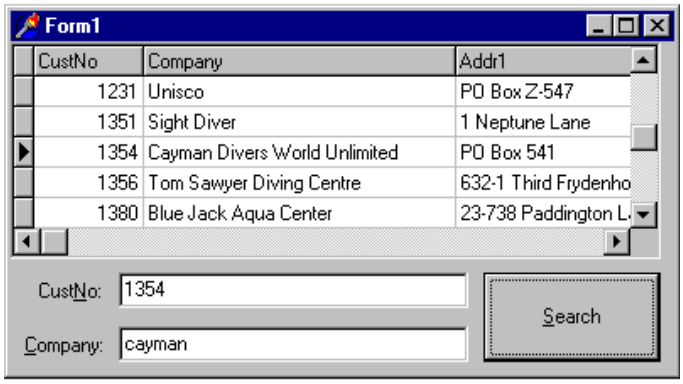


➤ *Figure 3*



➤ *Figure 4*

➤ *Figure 5*




```
procedure TForm1.FormCreate(Sender: TObject);
var Loop: Integer;
begin
  if FormStyle = fsStayOnTop then
    with DBLookupCombo1 do
      for Loop := 0 to ComponentCount - 1 do
        if Components[Loop] is TPopupGrid then
          with TPopupGrid(Components[Loop]) do
            SetWindowPos(Handle, HWnd_TopMost,
              0, 0, 0, 0, swp_NoMove or swp_NoSize)
end;
```

➤ *Listing 3*

Delphi 1 Stay On Top

Q I have a stay-on-top form with a `TDBLookupCombo` on it. When I try and drop its listbox part down, nothing happens. When the form's `FormStyle` is set back to `fsNormal` all works well again. Is this a known problem?

A Yes, this is a known bug. Delphi 2 users don't tend to use `TDBLookupCombo` controls: they use the new `TDBLookupComboBox` components. The problem occurs because the `TDBLookupCombo` is a fake combobox made from an edit and a listbox (`TPopupGrid` actually). Since the listbox is a separate window, not a child of the form (to allow it to be displayed partially outside the form), when the form is set to stay on top the listbox gets obscured by it (see Figure 3).

To fix it, you need to iterate through `TDBLookupCombo`'s `Components` array until you find a `TPopupGrid`. When you do, use its window handle and call `SetWindowPos` to make it also a stay on top listbox. Listing 3 shows the `OnCreate` handler from the DBCOMBO.DPR project on this month's disk. This turns Figure 3 into Figure 4.

Delphi 32 Locate Method

Q I am having trouble calling `Locate` on a `TTable` in Delphi 2. It is fine when trying to search for something with a simple single-field search, but I cannot see how to use it to search for values from more than one field at a time.

A `Locate` takes three parameters as shown in this declaration from the Delphi help:

```
function Locate(
  const KeyFields: string;
  const KeyValues: Variant;
  Options: TLocateOptions):
  Boolean;
```

The first parameter is a semicolon separated list of field names and the second parameter is a variant designed to take field values to search for. If there is more than one field, then the parameter must represent a variant array. This is easy to set up using the `VarArrayOf` function, which is designed to manufacture a one-dimensional variant array. A sample program that calls `Locate` on two fields of the `Customer` table is supplied as LOCATE.DPR (see Figure 5). The `Search` button searches for the record with the specified customer number and company name using the code in Listing 4 (note that a partial company can be entered).

```
if not Table1.Locate(
  'CustNo;Company',
  VarArrayOf([EdtCustNo.Text,
  EdtCompany.Text]),
  [loPartialKey,
  loCaseInsensitive]) then
  raise Exception.Create(
    'Search values not found')
```

➤ *Listing 4*

Win32 Help

Q I know I can generate a full-text search database for my Win32 API help file, but is there any other way of making the information more accessible for general browsing? I recall the Delphi 1 Windows API help file had a useful Overviews topic (amongst others) that showed many links to other useful areas in the help file.

A The 16-bit API help file had many useful topics that could be found by starting at the contents page. It was easy to get to the individual 16-bit help files' contents pages as there was an ever-present Contents button. The best approach with the 32-bit help files is to load them up manually, rather than accessing them through the help system. Use Windows Explorer and navigate to Delphi's HELP subdirectory. Double-click on WIN32.HLP and you are presented with its contents page showing lots of interesting categories.

ShowWindow Problem

Q I have a need for my application to manipulate other applications. Occasionally the other application is written in Delphi and it is here that I find my problem. I sometimes need to activate a window in the target app, which I do with ShowWindow. If the target app is minimised and I show the main form window it does get restored but the minimise button no longer functions.

A This goes back to the perennial issue of Delphi applications having an additional window (maintained by the Application object) that does lots of special things to make a Delphi application act generally Windows-like. However, because this window exists, certain API operations need to take special care when talking directly to forms. In short, ShowWindow is not enough.

When a Delphi application is minimised by minimising the main form, all forms that are open are simply hidden. The icon that is displayed to represent the application is in fact an iconic view of the Application window. When any other form is minimised it acts normally.

If another app needs to activate the main form then it must first restore the application from its minimised state. This can be done by sending a wm_SysCommand message to the Application window with a parameter of sc_Restore. It is always desirable to come up with some general solution to a problem, so Listing 5 has a procedure called ActivateWindow that takes a window caption and a window class name (and shows some sample calls to it). The function endeavours to activate the specified window, taking into account that it might be minimised and also that it might be in a Delphi application.

Notice that FindWindow is used to locate the target window. Delphi forms are not child windows, they are top level popup windows but which have a parent window (the Application window). Since FindWindow only works on top level windows (not child windows) we know that if it has a parent there is a good chance that it is a Delphi form. This rule isn't exactly foolproof, but generally non-Delphi apps make parent-less top level windows.

If a parent window is found, it is restored. The target window is also restored and brought to the front. This should work in all cases (I did some reasonably thorough testing), but I found one limitation with it. If the code is compiled in a Delphi 1 app and is used to activate a non-main form in a minimised 32-bit Delphi app, the form will be brought to the foreground, but will not have the focus. Focus is left on the main form. This is despite the Windows API help saying that BringWindowToTop "activates pop-up, top-level, and MDI child windows."

Projects APP1.DPR and APP2.DPR on the disk show how this routine is used. APP1 activates the two forms in APP2 as well as activating Excel if found.

Hi MOM!

Q I am trying to write a little project similar to the Microsoft Office Manager that launches applications. It's basically a form with a few speed buttons that run Word, Excel etc. I need to check if the application

➤ *Listing 5*

```
procedure ActivateWindow(Caption, ClassName: PChar);
var FormWnd, AppWnd: HWnd;
begin
  FormWnd := FindWindow(ClassName, Caption);
  if FormWnd = 0 then
    raise Exception.Create('Cannot find window');
{$ifdef Win32}
  { Having found the form, now find the Application window }
  { Can't reliably use GetParent, so... }
  AppWnd := GetWindowLong(FormWnd, gwl_HWndParent);
  { Tell the Delphi Application window to pop up in case it }
  { is minimised. Bear in mind that FindWindow only works on }
  { top-level windows, not child windows. Delphi forms are }
  { top-level popup windows which have parents, so the following }
  { check should only try and restore a parent window if it }
  { is a Delphi app }
  if (AppWnd <> HWnd_Desktop) and IsIconic(AppWnd) then
    SendMessage(AppWnd, wm_SysCommand, sc_Restore, 0);
  { Tell the form window to pop up if it is minimised }
  if IsIconic(FormWnd) then
    SendMessage(FormWnd, wm_SysCommand, sc_Restore, 0);
  { Make the target form be in front and active }
  SetForegroundWindow(FormWnd);
{$else}
  AppWnd := GetWindowWord(FormWnd, gww_HWndParent);
  if (AppWnd <> HWnd_Desktop) and IsIconic(AppWnd) then
    SendMessage(AppWnd, wm_SysCommand, sc_Restore, 0);
  if IsIconic(FormWnd) then
    SendMessage(FormWnd, wm_SysCommand, sc_Restore, 0);
  BringWindowToTop(FormWnd);
{$endif}
end;
...
ActivateWindow('MainForm', 'TMainForm');
...
ActivateWindow('SecondaryForm', 'TSecondaryForm');
...
ActivateWindow('Microsoft Excel', 'XLMAIN');
```

(eg Excel) is already running and if it is switch over to it. If it is not running I need to start the app. I cannot find how to test that Excel is running and then switch to it.

It seems like half of the answer to this question has already been dealt with in the previous section (*ShowWindow Problem*). The idea is to try and find a window in the target app that has certain known attributes. When you search for a window (typically using `FindWindow`) you can specify a window caption and/or a window class name to search for. The trouble with window captions is that they tend to get modified during a program's lifetime. Excel's main window might start out with the caption `Microsoft Excel` but it might change to `Microsoft Excel - Sheet1`. Windows class names are constant, however.

So we can search for a window class name and possibly check the caption if we know that some of it might be constant. If the window is not found, we launch the application. The `ActivateWindow` function from the previous section does the searching and activation if found. If it does not find the target window, it raises an exception. Note that if you wish to use `FindWindow` to search for a class name but no particular caption, then you must pass `nil` as the value for the `PChar Caption` parameter, *not* a pointer to an empty string. This is why the `ActivateWindow` routine takes unfriendly `PChar` parameters instead of nice strings.

We have a remaining question here though. How do we know what window class name is used by an application? The answer is to use WinSight, which lists all the windows that currently exist and gives details on their positions, class names, captions, parents and children (see Figure 6). WinSight tells me that my copy of Excel 3 has a main window class name of `XLMAIN` where Word 6 has `OpusApp` (these may well vary from version to version, although it wouldn't take an awful lot of research to get a comprehensive list for all versions). In the case of Delphi written apps, do not go for `TApplication` as a class name, since every Delphi app has a window of that class. Try another class name that will hopefully be unique.

So the speed buttons in the application could have code rather like this:

```
RunOrSwitchToApp(
  'c:\excel\excel.exe',
  'XLMAIN')
```

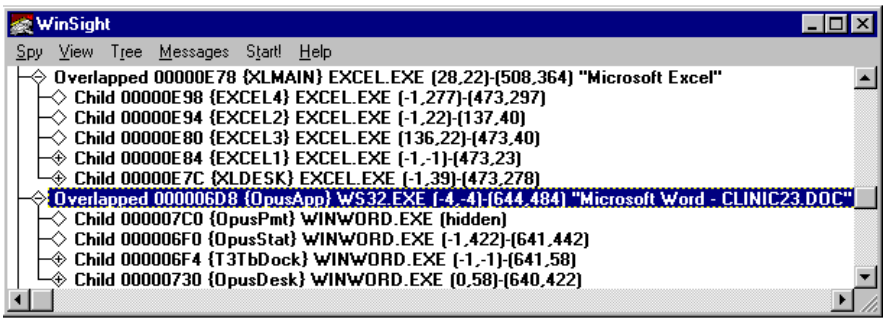
where `RunOrSwitchToApp` is the straightforward procedure in Listing 6. `ActivateWindow` we have seen before and `RunApp` simply calls `CreateProcess` or `WinExec` dependant upon platform. That leaves us with an application that is not a million miles away from the Microsoft Office Manager (or MOM): see Figure 7. Obviously this is a simple example, you wouldn't want to hardcode paths in your own applications.

Update: Executing Methods By Name

Issue 21's entry *Poor Person's Polymorphism* showed one way of executing arbitrary routines dependant upon some (ordinal) value. It was suggested that typically you could not take a string and call the procedure whose name it held because procedure names are stripped out by the compiler. This is generally true,

➤ *Figure 7*

➤ *Figure 6*



➤ *Listing 6*

```
procedure RunOrSwitchToApp(
  const AppPath: String;
  ClassName: PChar);
begin
  try
    ActivateWindow(nil, ClassName)
  except
    RunApp(AppPath)
  end
end;
```

➤ *Listing 7*

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
  public
  published
    procedure MyRoutine(const Msg: string);
  end;
  TCustomProc = procedure (const Msg: string) of object;
...
procedure TForm1.MyRoutine(const Msg: string);
begin
  MessageDlg(Msg, mtInformation, [mbOk], 0)
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('A button was pushed... or was it?')
end;
procedure TForm1.Button2Click(Sender: TObject);
var Proc: TNotifyEvent;
begin
  @Proc := MethodAddress('Button1Click');
  if Assigned(Proc) then
    Proc(nil);
end;
procedure TForm1.Button3Click(Sender: TObject);
var Proc: TCustomProc;
begin
  @Proc := Self.MethodAddress('MyRoutine');
  if Assigned(Proc) then
    Proc('Hello world');
end;
```

but there is an exception for published methods. RTTI (run-time type information) is generated for anything in a class's `published` section so that your application can use the textual references in the DFM file and map back to real classes, properties and methods (event handlers). Remember that the untitled section of a class that Delphi maintains has the same attributes as a `published` section.

With this information, we can take advantage of it ourselves. If we wish to execute a published method, given its name in a string, we can use the `TObject` method `MethodAddress` to give us its address. All that is then required is to typecast the returned pointer to an appropriate method pointer type and we can invoke it successfully.

All event handlers generated by the Object Inspector are `published` methods, but you can also place your own methods in a form's `published` section. Don't use the unnamed section that Delphi uses: make your own with the `published` keyword. Listing 7 shows some code from the sample CALLER.DPR project.

Acknowledgements

Thanks to John O'Connell for help with parts of this month's column.