The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?
Just email Brian Long, our Delphi Clinic Editor, on blong@compuserve.com or write/fax us at The Delphi Magazine

Window Startup Mode

Q If I make a shortcut to a Delphi 2 app and tell it to start minimised, the program completely ignores that flag and starts normally. There was a reasonably well-known fix for this for Delphi 1 apps. It fails to work in Delphi 2.

A When Windows starts an application it passes a flag along to it, indicating how the main window should display. Delphi takes no notice of this flag. For Delphi 1 apps, the solution was to check this flag (stored in the `System` unit variable `CmdShow`) and set the main form's `WindowState` property accordingly in its `OnCreate` handler (see Listing 1).

Delphi 2 is a bit more destructive when it comes to `CmdShow`: the `System` unit startup code explicitly sets `CmdShow` to be one particular value (`sw_ShowNormal`, or 10) thereby making Listing 1 ineffective (though apparently this will be fixed in Delphi 3). We have to be a bit more cunning about retrieving the original value which was passed in by calling a suitable Win32 API (`GetStartupInfo`). Having done that we then proceed as before. But then another problem comes our way.

It was reported in a Delphi Clinic entry back in Issue 9 that Delphi 2 had a problem with minimised forms. When a form was minimised, it doesn't minimise into the task bar but onto (above) the task bar. I showed some code supplied by Borland's Roy Nelson that showed how to get the main form to minimise correctly, but didn't explain at the time why the "problem" occurs in the first place.

Any multi-form Delphi 1 application (including Delphi 1 itself) has one icon in the taskbar for each form. This upset many people (because other applications don't do this) and so Delphi 2 has one icon on the taskbar representing the whole application. Bearing this in mind, consider what happens if a secondary form gets minimised. It would simply disappear with no way of bringing it back. To avoid this issue, the Delphi 2 designers ensured that minimised forms get placed above the taskbar in a manner similar to minimised Windows 95 MDI children.

But this minimised behaviour is also applied to the main form, which looks a bit silly. If the application is intended to start minimised then we'd expect to see an entry on the taskbar and nothing more. Listing 2 shows the relevant code to add to a Delphi 2 application to fix the minimised main form problem and simultaneously fix the

➤ *Listing 1: Starting a Delphi 1 app as Windows intended*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if WindowState = wsNormal then
    case CmdShow of
      sw_ShowNormal, sw_ShowNoActivate, sw_Show,
      sw_ShowNA, sw_Restore: WindowState := wsNormal;
      sw_ShowMinimized, sw_Minimize,
      sw_ShowMinNoActive: WindowState := wsMinimized;
      sw_ShowMaximized: WindowState := wsMaximized;
    end;
end;
```

➤ *Listing 2: Delphi 2 requires more that Delphi 1 does*

```
  TForm1 = class(TForm)
    ...
    procedure DoRestore(Sender: TObject);
  end;
...
procedure TForm1.DoRestore(Sender: TObject);
begin
  Application.ShowMainForm := True;
  { Restore application }
  Perform(wm_SysCommand, sc_Restore, 0);
  { This is needed to ensure all components draw properly }
  Show;
  { Disconnect the event handler so this code will not be called again }
  Application.OnRestore := nil;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  if WindowState = wsNormal then
    case CmdShow of
      sw_ShowNormal, sw_ShowNoActivate, sw_Show,
      sw_ShowNA, sw_Restore: WindowState := wsNormal;
      sw_ShowMinimized, sw_Minimize,
      sw_ShowMinNoActive: WindowState := wsMinimized;
      sw_ShowMaximized: WindowState := wsMaximized;
    end;
  if WindowState = wsMinimized then begin
    Application.ShowMainForm := False;
    Application.OnRestore := DoRestore;
    Application.Minimize;
  end;
end;
var StartupInfo : TStartupInfo;
initialization
  { Set up CmdShow correctly to workaround the fact that Delphi hardwires it to a
value 10 (sw_ShowNormal) }
  GetStartupInfo(StartupInfo);
  CmdShow := StartupInfo.wShowWindow;
end.
```

problem highlighted in the question. Again, this code comes from Roy Nelson who works in Borland's European Technical Team.

Note that some other solutions that get bandied about are a bit simpler than this and simply call `ShowWindow(Handle, CmdShow)` once `CmdShow` has its proper value, however this leaves the state of the window and the `WindowState` property inconsistent.

```
procedure TMainForm.Button2Click(Sender: TObject);
type
  TDllFunc = function: HResult; stdcall;
var
  DLL: THandle;
  RegFunc, UnregFunc: TDllFunc;
begin
  DLL := LoadLibrary('CFX32.OCX');
  if DLL = 0 then
    raise Exception.Create('OCX not found');
  @RegFunc := GetProcAddress(DLL, 'DllRegisterServer');
  @UnregFunc := GetProcAddress(DLL, 'DllUnregisterServer');
  try
    if @RegFunc = nil then
      raise Exception.Create('Registration routine not found');
    RegFunc;
    with TOCXForm.Create(Application) do
      try
        ShowModal;
      finally
        Free
      end;
    if @UnregFunc = nil then
      raise Exception.Create('Unregistration routine not found');
    UnregFunc;
  finally
    FreeLibrary(DLL)
  end;
  Close
end;
```

➤ *Listing 3*

OCX Deployment

Q I bought an OCX control recently. When I tried running an application that used it on another PC I got an `EOleError` exception saying that a class was not registered. What haven't I done?

A The OCX is a special version of an in-process OLE server: it's a DLL with some special stuff in. OLE servers need to be registered in the Windows registry and your OCX hasn't been. If you use an installation program to deploy your program, get it to load the OCX with `LoadLibrary` and call its `DllRegisterServer` function to do this. If you need to tidy the registry up later, you can call the OCX's `DllUnregisterServer` function.

The project OCX.DPR on the disk has two buttons on its main form, both of which load up a second form with the `ChartFX` OCX on it. The first button simply does a `ShowModal` operation, but the second one manually loads the OCX, registers it, calls `ShowModal` and then un-registers and unloads it. This should be enough to show you the necessary techniques. Listing 3 shows the button's event handler. To test the program, compile it and then copy the EXE and the `ChartFX` OCX (CFX32.OCX which is in the Windows\System directory) onto another machine. When you run the program one button will give the exception and the other will work fine due to the registration.

Oracle Packages

Q We are using Delphi 1 with Oracle 7.2 and I have trouble getting access to procedures stored in what Oracle calls "packages." I can execute a procedure in a package, using

```
begin
  package_name.procedure_name(
    .....);
end;
```

in a `TQuery` object, but I can't get at return parameters that this procedure may produce (they always seem to be set to zero).

A You should be able to use a `TStoredProc` to do this: the procedure name won't be available in the `StoredProcName` drop-down list but you can type it in using this syntax:

```
<ownername>.<packagename>.<PROCEDURENAME>
```

(case is important). However the problem is that it doesn't work. Borland have this logged as a bug. When you type the name in as above, what should be passed to Oracle is something like

```
"SCOTT"."MYPACK"."DUMMY"
```

but what the BDE erroneously passes is

```
"SCOTT"."MYPACK.DUMMY"
```

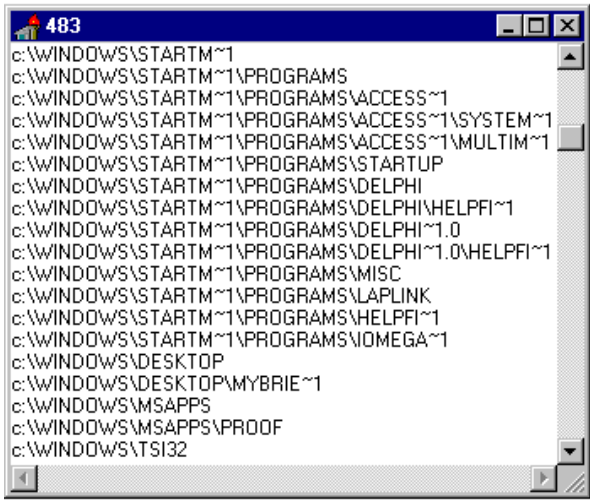
It gets the quotes wrong. Bad luck. However, on the brighter side, Borland's brand new SQL Links 3.5 appears to remedy the problem. I recommend upgrading.

File Finding

Q How do I iterate through all my subdirectories performing some arbitrary operation of my choice for each one?

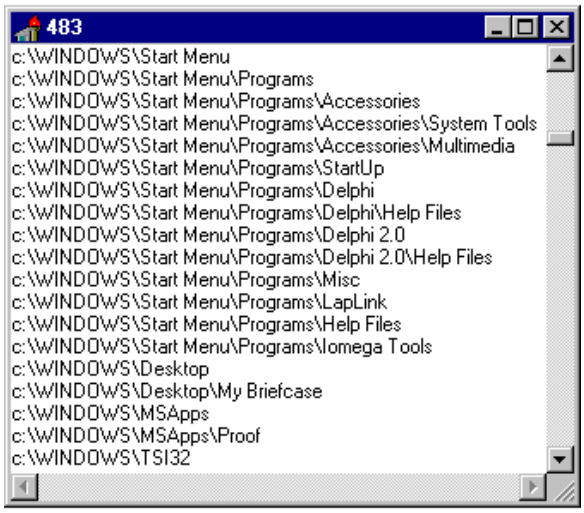
A This relies on using `FindFirst`, `FindNext`, `FindClose` and a `TSearchRec` record. Listing 4 shows a function that iterates around the directory tree (using recursion to go down through the subdirectories) and calls a supplied routine for each one. The callback must be a method that has the same interface as the one shown in Listing 5. The project FILELIST.DPR on the disk uses this routine to display all the directories on drive C: in a memo, and show a count of them on the form's caption bar. The iteration is started by double-clicking the memo (Listing 5 again). For speed of execution the memo is not continuously updated, but filled in when the iteration has finished. This is achieved by calling the `BeginUpdate` and `EndUpdate` methods of the `TStrings` object (`Lines` property) in the memo.

Figures 1 and 2 show the program having run through my hard disk when compiled in Delphi 1 and 2. You can see that the Win32



➤ *Left: Figure 1, Delphi 1 version*

➤ *Right: Figure 2, Delphi 2 version*




```
procedure GetSubDirs(const Dir: String; CallBack: TCallBack);
var SearchRec: TSearchRec;
    ThisDir: String;
begin
  if FindFirst(Dir + '\*.*', faDirectory, SearchRec) = 0 then
    try
      repeat
        { Only want directories }
        if (SearchRec.Attr and faDirectory <> 0) and
          { Don't want current or parent directory }
          (SearchRec.Name[1] <> '.') then begin
          ThisDir := Dir + '\' + SearchRec.Name;
          if Assigned(CallBack) then
            CallBack(ThisDir);
          GetSubDirs(ThisDir, CallBack);
        end;
      until (FindNext(SearchRec) <> 0) or Application.Terminated;
    finally
      FindClose(SearchRec);
    end;
end;
```

➤ *Listing 4*

```
procedure TForm1.MyCallBack(const Directory: String);
begin
  Memo1.Lines.Add(Directory);
  Caption := IntToStr(Memo1.Lines.Count);
  Application.ProcessMessages;
end;
procedure TForm1.Memo1DblClick(Sender: TObject);
begin
  Memo1.Lines.Clear;
  Memo1.Lines.BeginUpdate;
  GetSubDirs('c:', MyCallBack);
  Memo1.Lines.EndUpdate;
end;
```

➤ *Listing 5*

```
Self.Disable;
{ the Self bit is just in case
  you're in a with clause }
try
  {...do your processing...}
finally
  Application.ProcessMessages;
  { make sure the hardware events
    are taken by the disabled form }
  Self.Enabled := True;
end;
```

➤ *Listing 6*

executable gets to see all my long file names. Note that a routine for iterating through all directories on all drives was featured in the *Tips & Tricks* column in Issue 17. You should look at that for other ideas.

Eating Mouse Clicks And Key Presses

Q When I have some code executing in an event handler for several seconds, or a lengthy query executing, I set `Screen.Cursor` to `crHourglass`. This gives my users the idea that something is going on and that they should not go clicking all over my form. However, if they want to they can still click on things and those clicks get buffered up. When my code has finished the mouse clicks are actioned. How can I stop any stray clicks actually being received when the code finishes executing?

A There are 'a number of ways of approaching this problem. What SDK programmers used to do in the bad old days was to make a transparent child window of the form that would take all the hardware events during the busy processing and then remove it again when it was done. There is an article in an old *Microsoft Systems Journal* about doing it. These days, most people will be happy disabling the form and re-enabling it afterwards (see Listing 6).

Note the important use of `Application.ProcessMessages`. This must occur somewhere between the disabling and enabling of the form, otherwise the form will be back in its enabled state before any of the hardware messages leave the system message queue. This means that after the form is re-enabled, all the mouse clicks and keystrokes will be fed straight to it as before. The `ProcessMessages` call ensures that they arrive at the form when it is disabled and therefore will be ignored (or at the most cause a beep).

Acknowledgements

Thanks to Steve Axtell and Roy Nelson (both from Borland's European Technical Team) for help with some of this issue's entries.