Inside TApplication

by Nick Hodges

Most of the components used in building Delphi applications can be clearly seen on the Component Palette and manipulated with the Object Inspector at design time. A click on the palette and a click on the form, and any component on the palette is ready for use. However, the most important component to any Delphi application is not on the Component Palette, nor will its properties be found in the Object Inspector. `TApplication` is the foundation for all Delphi VCL based projects. It contains the lowest level of code needed to run a Windows application, creating the ever-patient message loop and handling all the low level calls to the Windows API that create and run an application. Like a Secret Service agent, `TApplication` is there, not quite noticed, but very capable and ready to serve.

Strangely enough, `TApplication` is actually a component, descending directly from `TComponent`. `TApplication` itself is declared in the `Forms` unit of the runtime library. The instance of `TApplication` that is declared for all Delphi projects, `Application`, is actually a Window, created directly with a call to the API `CreateWindow`. It is intialized with zero height and zero width, so it never actually appears on the screen. `Application` knows how to create and manage the main form of a Delphi project at runtime. `TApplication` has properties and events just like any other component. The best part is that a number of these events and properties contain valuable information for the Delphi programmer. That information is not readily apparent, but easily surfaced.

Application.ProcessMessages

Frequently, an application will have to perform a task that takes a rather large chunk of processor time. Often, this involves some sort of loop. Because Windows 3.x multi-tasks cooperatively, a well-behaved Windows application has to allow other applications a shot at processing their messages. `Application` provides a simple way to allow messages to be processed while a project is busy doing some other menial task. A call to `Application.ProcessMessages` anywhere in your code will ensure that your application will give other Windows programs space to do their thing. Periodic calls inside a loop will allow all applications to process messages that would otherwise be bottled up.

The sample application (included on the disk and shown in action over the page) demonstrates how this works. When the *Waste Time* check box is selected, the demo continuously counts up and down from 0 to 100 and displays the status in a gauge (see Listing 1). The `repeat...until` loop would normally seize control of the Windows environment, not allowing any other applications access to the message queue. However, a simple call to `Application.ProcessMessages` in the middle of the loop causes the demo to peek into the message queue and process any messages waiting there. As a result, Windows can function normally despite a loop running continuously in the background.

However, note that `Application.ProcessMessages` will not close an application when the `wm_quit` message is encountered inside a loop. Therefore, the loop itself includes a check `Application.Terminated`. This ensures that `Application.ProcessMessages` actually processes all the waiting messages for the application before terminating the application. `Application.ProcessMessages` actually sets `Terminated` to true, but the programmer must explicitly check for it to allow it to be processed. To see this work, try commenting out the call in the `until` clause, run the program and notice that the program won't close until the *Waste Time* check box is deselected.

You can call `Application.ProcessMessages` anywhere at any time, but it is best used when any action a program takes might interfere with the free flow of Windows messages. Interestingly, the code is the same as that invoked by `TApplication` when it sets up the message loop and waits for user input in any Delphi program.

Starting Out Minimized

Employing a zero-sized window to run a Delphi application and to manage all of its associated forms causes the application to behave slightly differently to what might normally be expected.

Despite how it may appear to a developer within Delphi itself, the real main window of any Delphi application is the `TApplication` window itself. It is this window which is displayed when the application is minimized and it is this window which is queried by Windows

➤ *Listing 1*

```
procedure TForm1.CheckBox3Click(Sender: TObject);
var Increment: Integer;
begin
  Increment := 1;
  repeat {Waste time, but allow processing of Windows messages}
    Gauge1.Progress := Gauge1.Progress + Increment;
    if Gauge1.Progress = Gauge1.MaxValue then Increment := -1;
    if Gauge1.Progress = Gauge1.MinValue then Increment := 1;
    Application.ProcessMessages;
  until (not CheckBox3.Checked) or (Application.Terminated);
  Gauge1.Progress := 0;
end;
```

shells such as Program Manager when seeking an icon.

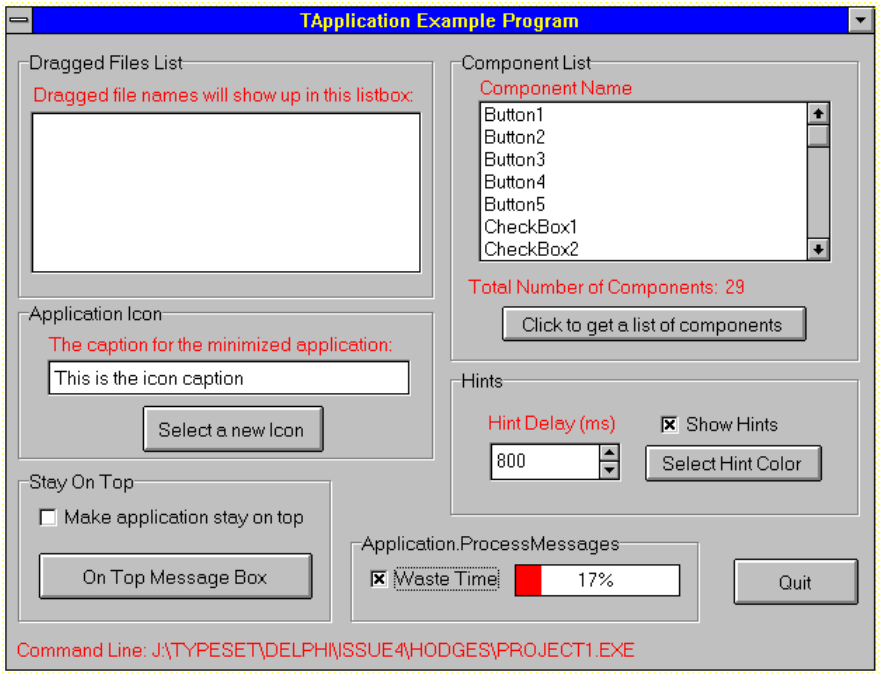
One of the easiest ways to show this slightly unusual trait is to create a simple Delphi application, install it in a group in Program Manager and then tell Program Manager to run the application minimized. The Delphi-built application will ignore the command when run from Program Manager. Since `Application` is really the main form of the application, and it creates and displays what the developer calls the main window of the application, the message never gets to the application to start in a minimized state. `TApplication` doesn't process the `CmdShow` parameter which defines how the program will be displayed on startup.

Fortunately, there is an easy fix to this seemingly anomalous behavior. The demo application, if started with the Run Minimized command set in Program Manager, will behave as expected. In the main form of the demo program, the `FormCreate` method checks the `CmdShow` value that was passed to `TApplication` and stored in the `CmdShow` variable in the `System` unit (see Listing 2). The `FormCreate` constructor checks the value and sets the `WindowState` accordingly. A call to the `ShowWindow` API would do the same thing, but wouldn't necessarily set the proper `WindowState` value for the main form.

Icon And Icon Caption

Some developers may notice that the once the application is properly minimized when called from Program Manager, the icon that is displayed in Program Manager is not the one attached to the main form's `Icon` property.

This is another symptom of the distinction between `TApplication` and the main form. The icon that is bound into the executable and found by Program Manager is the icon attached to the application itself. This icon can be set through Delphi's IDE on the `Options|Project|Application` page. You can also change the application's main icon at runtime with a simple assignment statement.



➤ *The example program in action*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  {Ensure Window opens itself in state set by CmdShow}
  case CmdShow of
    sw_ShowMinimized,
    sw_ShowMinNoActive     : WindowState := wsMinimized;
    sw_ShowMaximized       : WindowState := wsMaximized
  else
    WindowState := wsNormal
  end; { case }
  { ... more code here, see files on the disk ... }
end;
```

➤ *Listing 2*

The solution to this dilemma is to do one of two things: either ensure that the `TApplication` icon and the main form icon are the same, or leave the `Icon` property of the main form blank and let `TApplication` do all the icon management.

It is also easy to assume that the main form's caption will become the caption for the icon, but such is not the case. The `Title` property of `TApplication` holds a string that will be displayed as the minimized application's caption. The default can be set in the project's `Option` dialog, and can be easily changed at run-time.

The ever-present demo demonstrates the use of icons and their captions. Note that if placed in Program Manager, the demo will display the Delphi default icon. When run and minimized, that same icon will be displayed. However, `TApplication` has an `Icon` property that can be set at design time. The demo allows you to do that – see Listing 3. Assigning an Icon to the main form's `Icon` property at design time would cause that icon to be displayed on minimization, but not as the icon representing the application in Program Manager. Note, too, that the icon assigned at runtime is only temporary, and that the icon assigned to `TApplication` at design time is the one bound into the program at compile time as its main icon.

Finally, the caption can be easily changed by entering a string into the supplied edit box. That string is then assigned to `Application.Title`.

Dragged Files

The fact that the icon shown on minimization is not the icon

representing the project's main form brings about more unusual, but fixable, behavior in Delphi applications. Because the icon displayed when the program is minimized is owned by the application and not by the main form, dragging files from File Manager to the iconized application does not function as expected. You can cause an application's main form to accept dragged files as usual by calling the DragAcceptFiles API and responding to the wm_DropFiles message to gather information about those files. Once this is done, Delphi applications that are in the restored state will accept these files gladly; however, when minimized they will not.

TApplication has to be set up to accept files as well. TApplication has an event called OnMessage that is invoked every time a message is received by the application. By calling DragAcceptFiles and passing Application.Handle, and by writing a special handler to catch the wm_dropfiles message inside the OnMessage, a minimized application can respond to files dragged to it in exactly the same way as does a restored program. Note that the OnMessage event could be used to trap any Windows message that might need special handling by the Application instance, such as wm_paint for painting on the icon.

The demo application illustrates how a Delphi application can be set up to accept files in any state. Both TApplication and the program's main form are able to accept dragged files, and both respond to the wm_dropfiles message by gathering the names of all the dragged files in a TStringList and then placing that list into a listbox on the form.

Hints

Windows applications these days aren't considered complete without fly-by help boxes for buttons, tool bars and other components. Delphi makes it incredibly easy to supply these little hint boxes. The TApplication object makes it very easy to customize them. TApplication supplies properties to change the time a user waits to see the hints, whether the hints are displayed at all, and even the background color of the hints themselves, in case a programmer wants to be different and not display hints with the standard yellow background.

These features can be easily seen in the demo application. The hints can be turned on and off using the so-named check box. The hint delay time, in milliseconds, can be set using the spin edit box. Note that the delay is set only for the first hint, once the first hint is shown all hints after that are immediately displayed without delay. This allows users to see all the hints without having to wait for the delay for each control. Moving the mouse off the main window resets the delay. The background color of the hints can be changed with a simple call to a ColorDialog box. The selected color is then set to the TApplication HintColor property.

```
procedure TForm1.Button1Click(Sender: TObject);
var Icon: TIcon;
begin
  {Get an icon and load it into the Application.  The new icon will now
   show up when the application is minimized}
  Icon := TIcon.Create;
  if OpenDialog1.Execute then begin
    Icon.LoadFromFile(OpenDialog1.Filename);
    Application.Icon := Icon;
  end;
  Icon.Free;
end;
```

➤ *Listing 3*

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  Application.NormalizeTopMosts; {Allow dialogs on top}
  MessageBox(Form1.Handle, 'This should be on top.', 'Message Box', MB_OK);
  Application.RestoreTopMosts; {Return to normal}
end;
```

➤ *Listing 4*

Stay-On-Top

Some Delphi developers may want to create an application that always remains on top of all the other windows on the screen.

Interestingly, fsStayOnTop is the only FormStyle property setting that can be changed at runtime. However, when in this state, problems can arise when the application tries to call another dialog. Dialogs called by programs in stay-on-top mode can end up *behind* the calling window. If such a dialog is modal, it can lock up Windows entirely! TApplication allows you to place dialogs on top of forms that have fsStayOnTop set. The two methods NormalizeTopMosts and RestoreTopMosts allow a programmer to toggle in and out of a state that allows dialogs to be place on top of a stay-on-top application (see Listing 4).

The trusty demo application can be switched to stay-on-top mode and then can calls a message box which is displayed on top of the form. Without the calls altering the topmost state, the message box would be placed behind the app and out of reach, causing Windows to become modal with no escape. Even worse, users wouldn't even know what had happened! To demonstrate this, try setting the stay on top checkbox and then try to change the hint color!

Listing Components

Frequently, a Delphi developer may wish to gain access to a particular type of control or a certain set of controls on a form at run time. TApplication contains a list of all the components owned by the main form in its Components property. ComponentCount contains the total number of controls in the array. By using run-time typing information and a for loop, a programmer can cycle through all of the main form's components and

find components that are of a certain type or that have certain properties. The demo shows this by gathering all of the names of the components on a form and putting them in a list box (see Listing 5). It also picks out all the TLabel controls and alternates their font color between red and black. You can use this technique to find any specific control or type of control.

The Command Line

TApplication also stores details at run time about the command line used to run itself. The EXEName property stores the full path of the executable, which can be broken down using functions from SysUtils, such as ExtractFilePath and ExtractFileName. The demo displays its command line in a label upon execution.

Conclusion

TApplication can perform a number of other tricks, including restoring and minimizing itself as well as making it easy for to invoke a program's help file. Despite being hidden away, TApplication has a wealth of capabilities. Knowing a few tricks we can take advantage of the strengths of TApplication and overcome its few quirks.

Nick Hodges, an experienced Delphi and Pascal developer, is known to many as the author of TSmiley and the inspiration behind a whole raft of Smiley-Ware! He can be contacted via CompuServe on 71563,2250

Several readers responded to the query in Issue 3's Delphi Clinic about how to make an iconised Delphi application stay on top of all other windows – thanks! Here Hallvard Vassbotn provides a usefully generalised solution, as well as revealing other useful facets of TApplication.

Icon On Top

For a Delphi application, even if the main form's FormStyle property is set to fsStayOnTop, the icon which shows when minimized is not on top of the other windows. The reason for this is that the Application object maintains a hidden window that is the actual main window of the application. This hidden window will distribute commands to what Delphi considers the main form as it sees fit. When the Delphi main form is minimized, it will actually be *hidden* and the Application window is responsible for drawing the main form's icon.

With that in mind, and remembering that the Application window's handle can be accessed with it's Handle property, we can solve the problem using the code in Listing 6.

We simply hook the OnMinimize event of the Application object. Whenever the application is minimized, and thus the icon is showed, the code in AppMinimize will be run. Here we check if the FormStyle property of the main form indicates that the icon should be made on top. If so we use the WinProcs routine called SetWindowPos to change the display attributes of the icon.

Tile And Cascade

You might have noticed that when running an application created with Delphi it doesn't respond properly to the Tile and Cascade commands from the Task Manager. Delphi itself has this behaviour (it was, after all, written in Delphi!).

You can test this by running a Delphi app together with one or more non-Delphi apps. Bring up the Task Manager by double-clicking on the background or pressing Ctrl+Esc. Click the Tile and Cascade buttons. All non-Delphi applications are resized and positioned correctly. The Delphi app doesn't move, but instead an empty square is left where the window should have been placed.

This is another effect of the fact that the Application object in Delphi maintains its own hidden window which is the actual main window in Windows terms. The blank space you see when tiling is actually this hidden window.

To overcome this problem, we can use a little known feature of the Application object, the method HookMainWindow, which lets us hook into the message handler of the Application window. This way we can monitor and override any functionality of the main window.

By using the WinSight utility provided with Delphi, I found that monitoring WM_WindowPosChanging messages sent by Windows to the Application window would let me resize the main form correctly when tiling and cascading. The solution is shown in Listing 7.

First we hook the message handler of the application window with the HookMainWindow method. Note that Application keeps track of a list of hooks, so that there might be several hooks installed at once. When the main form is destroyed we act politely and clean up after ourselves by calling UnHookMainWindow.

The HookProc method will now be called for every message that arrives in the Application window's message queue. We are only interested in monitoring the messages, not overriding the

➤ *Listing 5*

```
procedure TForm1.Button2Click(Sender: TObject);
var I: Integer;
begin
  ListBox2.Clear;
  for I := 0 to ComponentCount - 1 do begin
    Listbox2.Items.Add(Components[I].Name);
    if Components[I] is TLabel then begin
      {Use Run-time typing to check type. Toggle the text of only TLabels
       between Red and Black. Note that each component must be typecast
       first.}
      if TLabel(Components[I]).Font.Color = clBlack then
        TLabel(Components[I]).Font.Color := clRed
      else
        TLabel(Components[I]).Font.Color := clBlack;
    end
  end;
  Label7.Caption := IntToStr(ComponentCount);
end;
```

➤ *Listing 6*

```
unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private   { Private declarations }
    procedure AppMinimize(Sender: TObject);
  public    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMinimize := AppMinimize;
end;
procedure TForm1.AppMinimize(Sender: TObject);
begin
  if FormStyle = fsStayOnTop then
    SetWindowPos(Application.Handle, HWnd_TopMost, 0, 0, 0, 0,
                 SWP_NoActivate or SWP_NoSize or SWP_NoMove);
end;
end.
```

➤ *Listing 7*

```
unit Unit2;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private   { Private declarations }
    function HookProc(var Message: TMessage): boolean;
  public    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.HookMainWindow(HookProc);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Application.UnHookMainWindow(HookProc);
end;
function TForm1.HookProc(var Message: TMessage): boolean;
var
  LocalFlags: word;
begin
  Result := false;
  if Message.Msg = WM_WindowPosChanging then begin
    with TWMWindowPosMsg(Message).WindowPos^ do begin
      if (hWnd = Application.Handle)
      and not IsIconic(hWnd)
      and (cx > 0) and (cy > 0) then begin
        LocalFlags := flags or SWP_NoZOrder;
        if BorderStyle = bsSizeable then
          LocalFlags := LocalFlags and not SWP_NoSize
        else
          LocalFlags := LocalFlags or SWP_NoSize;
        SetWindowPos(Self.Handle, 0, x, y, cx, cy, LocalFlags);
      end;
    end;
  end;
end;
end.
```

default behaviour, so we always return false.

If it is a `WM_WindowPosChanging` message, we are interested in it and type-cast the message record to the `TWMWindowPosMsg` defined in `Messages`. The `WindowPos` field is a pointer to a record that contains all the useful information, so we de-reference this pointer as well. Now to be on the safe side we check that the message was indeed intended for the `Application` window, that we are not an icon and that the size of the window is not zero.

If all is well so far, we know that we should resize the main form. To keep things unobstructed, we fiddle with the flag bits to make sure that the Z-order is not affected and that the size of a fixed-size window isn't changed.

Now Tile and Cascade from the Task Manager should work the way they are supposed to do. This code example also shows how to monitor and/or override the application window's behaviour – again demonstrating Delphi's power and extensibility!

Hallvard Vassbotn lives and works in Norway and can be reached by email at hallvard@falcon.no