Appbars And Clipboard Viewers

by Paul Warren

Have you ever needed to monitor the clipboard while you are doing copy and paste operations between applications? If you have, you've probably used the Windows clipboard viewer. Unfortunately, the clipboard viewer isn't the most convenient or flexible tool. First you have to remember to launch it, and then it always seems to be in the way.

If you need to have a clipboard viewer open, wouldn't it be nice if it could be made to 'stick' to the bottom of the screen, out of the way? It would also be nice if other applications were aware of it so that, when maximized, they don't cover, or get covered by, the viewer. In fact, wouldn't it be nice if the viewer behaved just like the Windows Taskbar? I certainly thought it would, so I did a little poking around in the Windows SDK files to see how it could be done.

It seems that the API includes methods for creating clipboard viewer windows and also Taskbar-like windows (called Appbars). Appbar applications in particular appear to have many potential uses. So let's get down to business and see how to create an Appbar.

What Is An Appbar?

If you look in the Windows SDK for *'Appbar Size and Position'* you will find a topic that is surprisingly lucid for Microsoft documentation. The topic states that:

'An application should set an Appbar's size and position so that it does not interfere with any other Appbars or the Taskbar. Every Appbar must be anchored to a particular edge of the screen, and multiple Appbars can be anchored to an edge. However, if an Appbar is anchored to the same edge as the Taskbar, the system ensures that the Taskbar is always on the outermost edge.'

This is exactly what I wanted for my proposed clipboard viewer. If you look for the topic *'Appbar Notification Messages'* in the SDK you will find additional information about how the system interacts with Appbars. The topic explains that the *'system sends messages to notify an Appbar about events that can affect its position and appearance. The messages are sent in the context of an application-defined message'*. The application-defined message is passed to the system by a call to `SHAppBarMessage(ABM_NEW, abd)` where abd is an `APPBARDATA` structure maintained by the window.

At its simplest, then, a window becomes an Appbar when it registers itself with `SHAppBarMessage(ABM_NEW, abd)`, responds to notification messages and unregisters itself with `SHAppBarMessage(ABM_REMOVE, abd)`.

The APPBARDATA Structure

As noted, our application needs to maintain an `APPBARDATA` structure. Although I could find no reference to it in Borland's documentation, the unit `ShellAPI` defines a `TAppBarData` type (along with the various constants we will need). Listing 1 shows the declaration taken from the `ShellAPI` unit.

`cbSize` is the size of the data structure and `hWnd` is the handle to the Appbar window. These two fields are needed every time you call `SHAppBarMessage`.

`uCallBackMessage` is the application-defined message. `uEdge` is a constant describing which edge of the screen we want to use. The possible values are `ABE_BOTTOM`, `ABE_LEFT`, `ABE_RIGHT` and `ABE_TOP`. `rc` is a `TRect` to pass bounding rectangles to and from the system. These last three fields are used when calling `SHAppBarMessage` with the values `ABM_NEW`, `ABM_QUERYPOS` and `ABM_SETPOS`. `lParam` is needed to pass specific information such as whether a full screen application is opening or closing.

A Basic Appbar Framework

Now we have enough background to create a basic Appbar framework. Create a new application and set the main form's `Height` property to the height you would like the Appbar to occupy on the bottom edge of the screen. This is to save adjusting the height programmatically. Set the `BorderStyle` property to `bsSizeToolWindow` and `FormStyle` to `fsStayOnTop`.

Now add a `Memo` component to the form and set `Align` to `alClient`. The `Memo` is in anticipation of making the application into a clipboard viewer later.

Next we need the application-defined message to pass to the system. I used the constant `wmAppbarMessage = wm_User` as the message constant. We also need a message handler to respond to the system generated notification messages.

Now add `abd: TAppBarData` to the `private` section of the form declaration to hold the application managed data.

In the `FormCreate` event handler we can initialize the `abd` structure and place our call to `SHAppBarMessage(ABM_NEW, abd)`. Set `abd.Size` equal to `SizeOf(abd)`,

Why Not Use Align

Some of you may be saying 'why go to all this trouble, you can set the `Align` property of a form to `alBottom` and get the same effect'. While it's true that setting the `Align` property this way does make the form look like an Appbar, it certainly doesn't behave like one.

First of all, the desktop doesn't rearrange itself as it does when an Appbar is created. Desktop icons may get covered up and maximized windows will be partially covered. Second, the form will get covered if you subsequently maximize a window. Finally, the forms don't 'stack' when you run several of them.

then abd.hWnd to our form's Handle and abd.uCallBackMessage to wm_AppBarMessage. All the other fields are ignored here, although for simplicity I have set them anyway.

According to the SDK an Appbar first *'proposes a screen edge and bounding rectangle for the Appbar by sending the* ABM_QUERYPOS *message'* and the system *'adjusts the rectangle (if necessary), and returns the adjusted rectangle to the application'*. At this point the application sends an ABM_SETPOS message. The system may adjust the proposed bounding rectangle again, so we use the adjusted rectangle in a call to SetBounds. We can propose and set the form's initial size and position in the FormCreate handler after the call to SHAppBarMessage(ABM_NEW, abd). In the FormDestroy handler we clean up with a call to SHAppBarMessage(ABM_REMOVE, abd).

Because the Appbar is frequently required to propose a bounding rectangle, we'll add a read-only property RequestRect to the form declaration and use a property access method to set RequestRect.

Finally, the SDK states *'Whenever an Appbar receives the* WM_ACTIVATE *message, it should send the* ABM_ACTIVATE *message. Similarly, when an appbar receives a* WM_WINDOWPOSCHANGED *message, it must call* ABM_WINDOWPOSCHANGED. *Sending these messages ensures that the system properly sets the Z order of any autohide appbars on the same edge.*' Thus we need a FormActivate event handler where we can send the ABM_ACTIVATE message and a WMWindowPosChanged message handler where we can send ABM_WINDOWPOSCHANGED. The application framework is shown in Listing 2.

If you were to run the application at this stage you'd find it would appear where it is supposed to, but it would not respond to changes in the Taskbar size or position. It can also be moved around the desktop freely, which is not correct Appbar behaviour. To address the first issue we need to complete the WMAppBarMessage handler.

Responding To Appbar Notification Messages

An Appbar may receive any of four messages. These are ABN_FULLSCREENAPP, ABN_POSCHANGED, ABN_STATECHANGE and ABN_WINDOWARRANGE.

ABN_STATECHANGE is designed to allow an Appbar to match state with the Taskbar when the user toggles the Stay On Top or Autohide features. ABN_WINDOWARRANGE notifies an Appbar when the user selects cascade or tile from the Taskbar context menu. For this example we won't need to respond to these messages.

We do need to respond to ABN_FULLSCREENAPP because, like the Taskbar, a well-behaved Appbar should hide itself when a full screen application is running. An Appbar will receive the ABN_FULLSCREENAPP message once when the first full screen app is started and once when the last full screen app is closed. The Msg.lParam field is true if the application is opening and false if it is closing. Therefore we should call Hide if Msg.lParam is True and call Show if it is not.

Finally, we need to respond to ABN_POSCHANGED that is called when an event has occurred that may affect the Appbar's size and position. These events include changes in the Taskbar's size,

➤ *Listing 1*

```
_AppBarData = record
  cbSize: DWORD;
  hWnd: HWND;
  uCallbackMessage: UINT;
  uEdge: UINT;
  rc: TRect;
  lParam: LPARAM; {message-specific}
end;
TAppBarData = _AppBarData;
```

➤ *Listing 2*

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, ShellAPI, StdCtrls;
const
  wm_AppBarMessage = wm_User;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  private
    abd: TAppBarData;
    function GetRequestRect: TRect;
    procedure WMAppBarMessage(var Msg: TMessage); message
      wm_AppBarMessage;
    procedure WMWindowPosChanged(var Msg: TWMWindowPosMsg);
      message WM_WindowPosChanged;
    property RequestRect: TRect read GetRequestRect;
  public
  end;
var
  Form1: TForm1;
implementation
var
  F: TextFile;
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  // fill the AppBarData data structure
  abd.cbSize := sizeof(abd);
  abd.hWnd := Handle;
  abd.uCallBackMessage := wm_AppBarMessage;
  abd.uEdge := ABE_BOTTOM;
  abd.rc := RequestRect;
  abd.lParam := 0;
  SHAppBarMessage(ABM_NEW, abd);
  // set the initial size and position
  SHAppBarMessage(ABM_QUERYPOS, abd);
  abd.rc.Top := abd.rc.Bottom - Height;
  SHAppBarMessage(ABM_SETPOS, abd);
  SetBounds(abd.rc.Left, abd.rc.Bottom - Height,
    abd.rc.Right-abd.rc.Left, Height);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  SHAppBarMessage(ABM_REMOVE, abd);
end;
procedure TForm1.FormActivate(Sender: TObject);
begin
  SHAppBarMessage(ABM_ACTIVATE, abd);
end;
function TForm1.GetRequestRect: TRect;
begin
  // set the requested Rect
  Result.Left := 0;
  Result.Top := Screen.Height - Height;
  Result.Right := Screen.Width;
  Result.Bottom := Screen.Height;
end;
procedure TForm1.WMAppBarMessage(var Msg: TMessage);
begin
end;
procedure TForm1.WMWindowPosChanged(var Msg:
TWMWindowPosMsg);
begin
  // must send this message to maintain correct Z-order
  SHAppBarMessage(ABM_WINDOWPOSCHANGED, abd);
  inherited;
end;
end.
```

position, and visibility state, as well as the addition, removal, or resizing of another Appbar on the same side of the screen. To behave correctly our Appbar must first set `abd.rc` to the proposed rectangle that is provided by the `RequestRect` property. Then we call `SHAppBarMessage(ABM_QUERYPOS, abd)`, adjust the rectangle and call `SHAppBarMessage(ABM_SETPOS, abd)`. Lastly, we call `SetBounds` with the adjusted rectangle. Listing 3 shows the completed message handler.

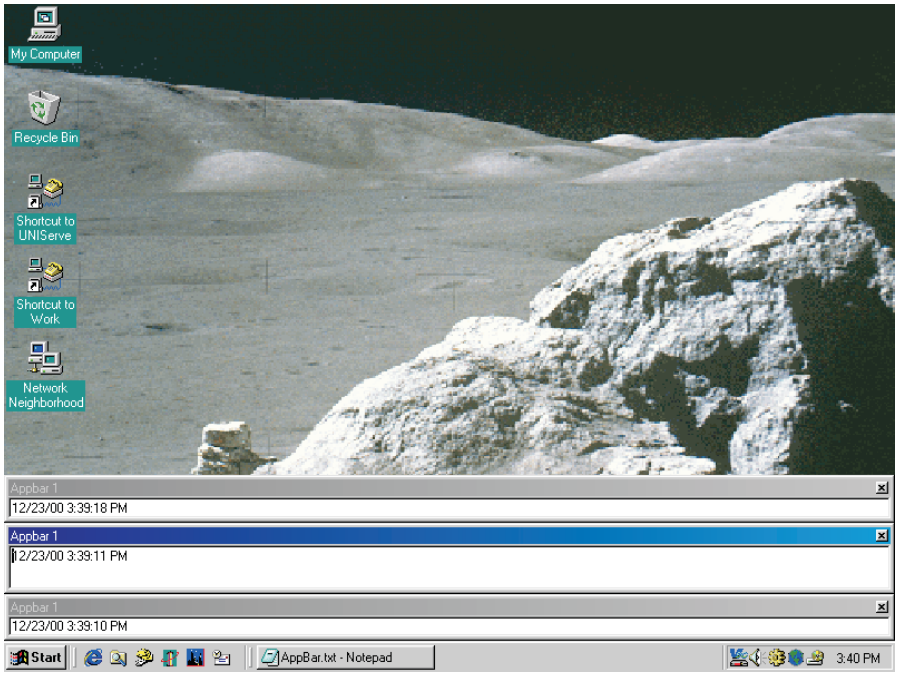
Our Appbar will now properly 'stick' to the bottom edge of the screen and respond to changes you make to the Taskbar or to other Appbars. You can still move the window away from its proper location by dragging it, though. So how do we fix this unwanted behavior?

Controlling The Sizing And Moving Rectangles

It is possible to set up a `WMWindowPosChanging` message handler to control an Appbar's size and position. In fact, this was the first method I tried to prevent the Appbar from moving. There is, however, a more elegant way. If we set up handlers for the `WM_SIZING` and `WM_MOVING` messages we can control the behavior of the Appbar, as well as display sizing and moving rectangles that only show legal positions.

Inside the `WMMoving` handler we set `PRect(Msg.lParam)^` to the current `abd.rc` value of the Appbar. This way any attempt to move the Appbar shows a fixed and unmoving rectangle in the current location. This indicates clearly to the user that movement of the window is not allowed.

➤ *Figure 1*




```
procedure TForm1.WMAppBarMessage(var Msg: TMessage);
begin
  // hide when fullscreen apps are displayed
  if Msg.wParam = ABN_FULLSCREENAPP then
    if Msg.lParam <> 0 then
      Hide
    else
      Show;
  if Msg.wParam = ABN_POSCHANGED then begin
    // fill the AppBarData data structure
    abd.rc := RequestRect;
    SHAppBarMessage(ABM_QUERYPOS, abd);
    abd.rc.Top := abd.rc.Bottom - Height;
    SHAppBarMessage(ABM_SETPOS, abd);
    SetBounds(abd.rc.Left, abd.rc.Bottom-Height,
      abd.rc.Right-abd.rc.Left, Height);
  end;
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
procedure TForm1.WMMoving(var Msg: TMessage);
begin
  PRect(Msg.lParam)^ := abd.rc;
  inherited;
end;
procedure TForm1.WMSizing(var Msg: TMessage);
begin
  PRect(Msg.lParam)^ :=
    Rect(Left, PRect(Msg.lParam)^.Top, Width, PRect(Msg.lParam)^.Bottom);
  inherited;
end;
procedure TForm1.WMExitSizeMove(var Msg: TMessage);
begin
  abd.rc.Top := abd.rc.Bottom - Height;
  SetBounds(abd.rc.Left, abd.rc.Top, abd.rc.Right-abd.rc.Left,
    abd.rc.Bottom-abd.rc.Top);
  SHAppBarMessage(ABM_SETPOS, abd);
  inherited;
end;
```

Similarly, inside the `WMSizing` handler we set `PRect(Msg.lParam)^` to a `Rect` computed from the current width and the new height. All that remains is to call `SetBounds` when the moving or sizing operation is complete.

We could try calling `SetBounds` in the `WindowPosChanged` handler, but it causes unwanted side effects because of the `WindowPosChanged` messages sent during the creation of the window. This causes the Appbar to appear in the wrong position.

A better place to call `SetBounds` is in response to a `WM_EXITSIZEMOVE` message. Inside a new message handler we can set `abd.rc.Top` to `abd.rc.Bottom-Height`, call `SetBounds` and send the `ABM_SETPOS` message needed to keep the registered Appbar bounds synchronized with the actual bounds. Listing 4 shows the new message handlers.

We now have a working Appbar that sticks to the bottom edge of the screen and properly responds to changes in the Taskbar size and position. Not only that, but you can launch several of these Appbars and they all respond to changes in

```
procedure TClipDialog.WMDrawClipboard(var Msg: TWMDrawClipboard);
begin
  Memo1.Text := Clipboard.AsText;
  SendMessage(CBHandle, WM_DrawClipboard, 0, 0);
  inherited;
end;
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
procedure TClipDialog.WMChangecbchain(var Msg: TWMChangecbchain);
begin
  if Msg.Remove <> CBHandle then
    SendMessage(Msg.Next, WM_Changecbchain, Msg.Remove, Msg.Next)
  else
    CBHandle := Msg.Next;
  inherited;
end;
```

```
function TForm1.GetRequestRect :
  TRect;
begin
  // set the requested Rect
  Result.Left := 0;
  Result.Top :=
    Screen.Height - Height;
  Result.Right := Screen.Width;
  Result.Bottom := Screen.Height;
  case abd.uEdge of
    ABE_TOP: begin
      // set the requested Rect
      Result.Left := 0;
      Result.Top := 0;
      Result.Right := Screen.Width;
      Result.Bottom := Height;
    end;
    ABE_LEFT: begin
      // set the requested Rect
      Result.Left := 0;
      Result.Top := 0;
      Result.Right := 50;
      Result.Bottom := Screen.Height;
    end;
    ABE_RIGHT: begin
      // set the requested Rect
      Result.Left := Screen.Width-50;
      Result.Top := 0;
      Result.Right := Screen.Width;
      Result.Bottom := Screen.Height;
    end;
  end;
end;
```

➤ *Listing 7*

the Taskbar and in each other. Figure 1 shows a number of Appbars in action. You will find this project on this month's disk as Project1.dpr. Now let's turn this Appbar into a clipboard viewer, before finishing up with some Appbar enhancements.

The Clipboard Viewer Chain

Turning our Appbar into a clipboard viewer is relatively easy. If you look under *Clipboard Viewer Windows* in the SDK you will find a short topic describing how clipboard viewers function. Basically, a viewer must add itself to the chain of clipboard viewers using a call to `SetClipboardViewer(Handle)`. The return value is the handle to the next viewer in the chain. We will put this call into the `FormCreate` event.

Having added our Appbar to the viewer chain we must respond to `WM_DrawClipboard` messages by first displaying the clipboard contents and then passing the message on to the next window in the chain. This is done with a call to `SendMessage`. Listing 5 shows the `WMDrawClipboard` handler. Note that this is a text-only clipboard viewer: if you want to display additional data types this is where you would make any changes.

Clipboard viewers must also respond to the `WM_ChangeCBChain` message that is sent whenever a window is removed from the chain. Like the `WM_DrawClipboard` message, `WM_ChangeCBChain` is sent to the first window in the clipboard viewer chain. Each window in the chain must pass the `WM_CHANGECBCHAIN` message to the next window in the chain, unless the next window is the window being removed. In this case, the viewer should save the handle specified by `hWndNext` as the next window in the chain. Listing 6 shows the message handler code for `WM_ChangeCBChain`.

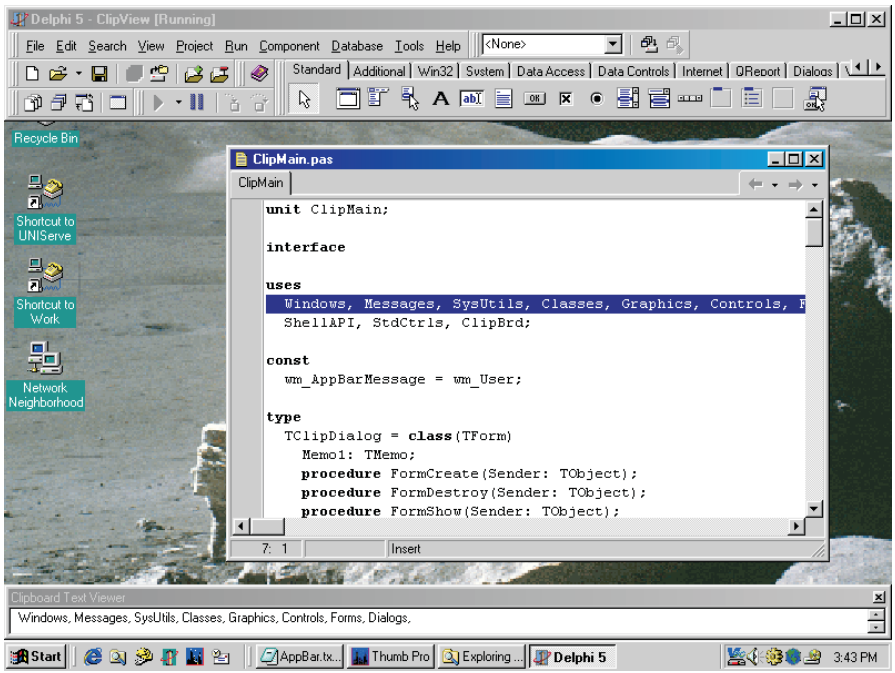
Finally, before closing, a clipboard viewer window must remove itself from the clipboard viewer chain by calling the `ChangeClipboardChain` function in the `FormDestroy` method.

So now we have a clipboard viewer window that is also an Appbar. You will find this project on the disk as ClipView.dpr. In Figure 2 you can see ClipView running. Now let's enhance the original Appbar by making it 'stick' to any edge of the desktop.

Enhanced Appbar

There are only a couple of changes needed to make our original Appbar capable of being moved to any edge of the screen. The first change is to modify the `GetRequestRect` method so that an appropriate bounding rectangle is returned according to the edge we want to attach to. Listing 7 shows the modified method. Note that this Appbar will not be resizable, I'll leave that enhancement to the interested reader.

➤ *Figure 2*



The next change is to modify the `WMMoving` message handler. Here we can change the drag rectangle displayed by the system so that it shows only the legal locations. Basically we divide the screen into four regions with the diagonals. If the cursor is in the left region we show the drag rectangle on the left edge and so on. Listing 8 shows the new message handler.

Finally, we can change the form's `BorderStyle` to `bsToolWindow` from `bsSizeToolWindow` since sizing is not allowed (more on `bsToolWindow` later). Project2.dpr on the disk creates an Appbar on the bottom edge of the screen and allows the user to drag it to any other edge. Figure 3 shows the Appbar and moving rectangle just before it moves to the right screen edge.

An AppToolBar

One potential use for an Appbar is to stick a Toolbar to a screen edge. The Toolbar would launch services for the user (just like the Microsoft Office Startup Toolbar).

To create an AppToolBar (this is my terminology by the way) all we need to do is to change the `BorderStyle` property of our form to `bsNone` and then change the `RequestRect` property to return the desired position.

When we run the new Appbar it appears where it should, but if you then resize the Taskbar our Appbar momentarily appears where it should and then jumps up by a distance equal to its own height. If we change the `BorderStyle` back to `bsToolWindow` it behaves correctly. What is going on here?

Whenever the Taskbar or an Appbar is moved or sized, Windows sends the notification

➤ *Listing 8*

```
procedure TForm1.WMMoving(var Msg: TMessage);
var
  P: TPoint;
begin
  GetCursorPos(P);
  if (P.Y < Screen.Height/Screen.Width*P.X) and
    (P.Y < -(Screen.Height/Screen.Width*P.X)+Screen.Height)
    then begin
    abd.uEdge := ABE_TOP;
    abd.rc := RequestRect;
    SHAppBarMessage(ABM_QUERYPOS, abd);
    PRect(Msg.lParam)^ := abd.rc;
  end;
  if (P.Y >= Screen.Height/Screen.Width*P.X) and
    (P.Y < -(Screen.Height/Screen.Width*P.X)+Screen.Height)
    then begin
    abd.uEdge := ABE_LEFT;
    abd.rc := RequestRect;
    SHAppBarMessage(ABM_QUERYPOS, abd);
    PRect(Msg.lParam)^ := abd.rc;
  end;
  if (P.Y >= Screen.Height/Screen.Width*P.X) and
    (P.Y >= -(Screen.Height/Screen.Width*P.X)+Screen.Height)
    then begin
    abd.uEdge := ABE_BOTTOM;
    abd.rc := RequestRect;
    SHAppBarMessage(ABM_QUERYPOS, abd);
    PRect(Msg.lParam)^ := abd.rc;
  end;
  if (P.Y < Screen.Height/Screen.Width*P.X) and
    (P.Y >= -(Screen.Height/Screen.Width*P.X)+Screen.Height)
    then begin
    abd.uEdge := ABE_RIGHT;
    abd.rc := RequestRect;
    SHAppBarMessage(ABM_QUERYPOS, abd);
    PRect(Msg.lParam)^ := abd.rc;
  end;
  inherited;
end;
```

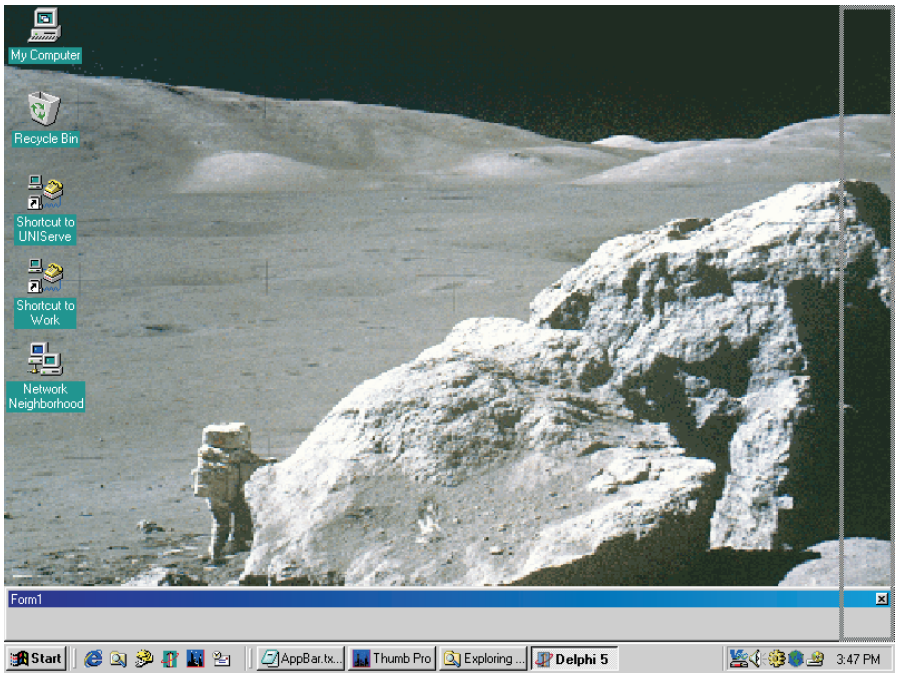
message to all other Appbars. It also sends messages to all top-level windows so that they can adjust themselves if necessary.

Our problem is caused by the hidden `Application` window. The main form (which is our Appbar) positions itself properly but then the application receives the `WM_WINDOWPOSCHANGING` message and passes it to our form. The form then repositions itself again. This time, however, it thinks that there is another Appbar present and therefore sets its position above where it should be. Windows which have their `ExStyle` set to `WS_EX_TOOLWINDOW` do not get the message and so do not behave the way they should.

The cure, then, is to set the `WS_EX_TOOLWINDOW` style in an overridden `CreateParams` method. This way, regardless of the `BorderStyle` which has been chosen, the form will be a `ToolWindow`. Listing 9 shows the code for the `CreateParams` method and Figure 4 shows the AppToolBar on my computer's desktop.

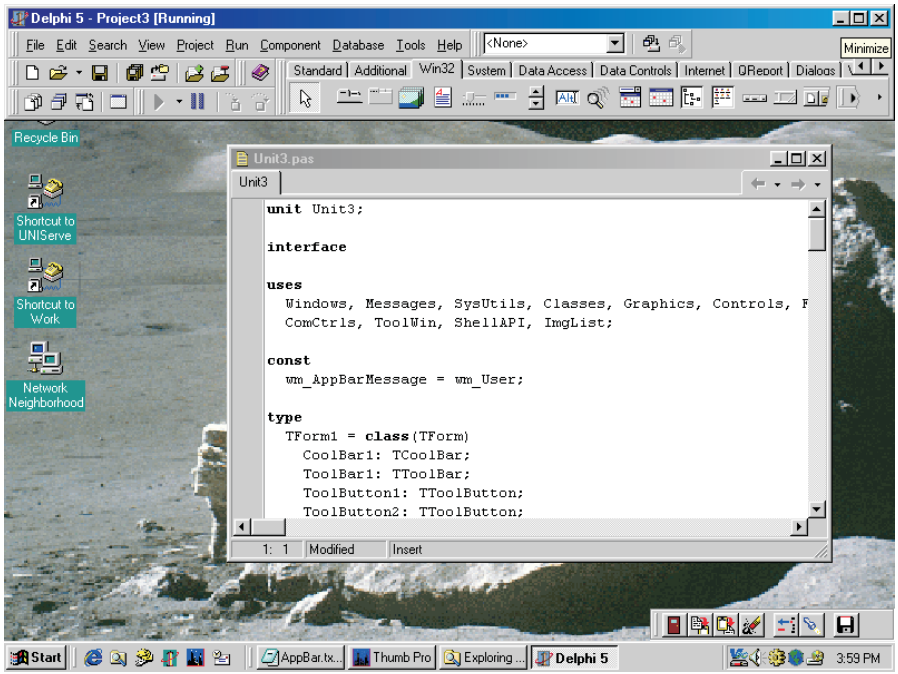
➤ *Listing 9*

```
procedure TForm1.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  if Params.ExStyle and WS_EX_TOOLWINDOW = 0 then
    Params.ExStyle := Params.ExStyle + WS_EX_TOOLWINDOW;
end;
```



➤ *Above: Figure 3*

➤ *Below: Figure 4*



Tidying Up

One last issue is the Appbar icon that appears on the Taskbar. In some cases, it seems appropriate to have the icon appear. For example, having the icon visible with the clipboard viewer application seems all right. With the Toolbar, though, it doesn't seem right at all.

The simplest way to hide the icon is to call `ShowWindow(Application.Handle, SW_HIDE)`. Apart from an almost unnoticeable flicker, this works just fine.

If the almost invisible flicker is unacceptable then you need to read *Windows 95 Tray Icons* by Marco Cantu in Issue 12 for another method of hiding the Taskbar icon.

Summary

Appbars seem to be an overlooked part of the Windows API. The ability to dock a form to the edge of the desktop strikes me as very useful. A clipboard viewer is just one tool I can think of that is more convenient when docked to a screen edge.

Even more interesting would be docking tear-away tool windows with the desktop. For example, imagine that you could dock the Delphi debug windows with the desktop as well as with the Delphi IDE windows. Although I haven't tried this yet, I can see no reason why it couldn't be done using the Appbar API.

Paul Warren runs HomeGrown Software Development in Langley, British Columbia, Canada, and can be contacted at hg_soft@uniserve.com