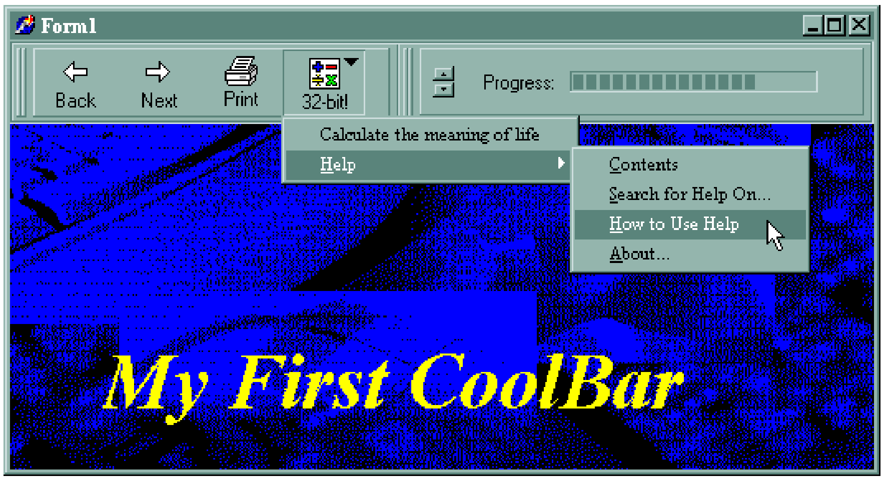
Beating the System: Explorer-Style Active Buttons

by Dave Jewell

Last month I promised to describe the development of an Internet Explorer style button control. I designed this button to complement the Coolbar code that I developed a couple of months back. Since then, I've taken the plunge and installed Office97 and it's now obvious that Microsoft are using the same combination of Coolbars and active button controls throughout all their new Office applications, setting a new standard for user interface design.



➤ *Figure 1: Here's a view of the TExplorerButton control being used in a demo application: the right-most of the four buttons has an attached popup menu which you can see being used*

So What's An Active Button?

When I use the term *active button control* or just *active button*, I'm not referring to ActiveX or anything of that nature. Rather, I'm using my own home-grown terminology for the way in which Microsoft's recently developed controls recognise that the mouse is moving over them (without pressing a mouse button) and change their appearance to indicate that they can be clicked. This 'sit up and beg' behaviour is an excellent idea because it provides immediate visual feedback to novice users, showing them what parts of a user interface will respond to mouse clicks.

Because of Delphi 3's ability to convert VCL components into ActiveX controls, and because of the introduction of Microsoft's Visual Basic 5.0 CCE (Control Creation Edition) development system, you can create custom ActiveX controls more easily than ever before. I anticipate an explosion of new controls over the next twelve to eighteen months and it's therefore critical to make your new control as easy to use and intuitive as possible. Making it 'active' goes a long way towards achieving this. At the same time, it's important to make your new controls consistent with industry leaders such as Microsoft. I've endeavoured to make this control respond as much like the Microsoft equivalent as possible.

Just to show you where we're going, take a look at Figure 1, which shows a small demo program that I knocked up. This demo uses the Coolbar code from a couple of months ago, with one of the Coolbar bands containing a `TPanel` component. Inside this panel are four of our new Explorer style buttons. As with Microsoft's buttons, `TExplorerButton` can display itself in one of four different ways: active (when the mouse is over the button), inactive (the button's normal 'idle' state), disabled, or pressed down (when the mouse is actually clicked over the button). In Figure 1, the fourth button is pressed down and displaying a drop-down menu.

As you move the mouse over a button, it automatically toggles from the inactive to active state. This is indicated by drawing a raised border around the button and changing the displayed glyph (Borland's term for small bitmaps displayed inside button controls) from a monochrome bitmap to a colour one. Similarly, when a button is disabled, the glyph is drawn with a greyed-out appearance and the button's caption text (if any) is drawn in a three-dimensional, greyed-out manner. It's worth pointing out that the caption and glyphs are optional: you can have both, you can have a button with a caption only, or you can just have the glyph. You can even have a button with neither caption nor glyph, but it doesn't look terribly exciting!

The Explorer button supports a number of different event types, with which you can associate an event handler in the normal Delphi way. In addition to standard events such as `OnClick` and `OnMouseMove`, there are a couple of custom events defined (`OnMouseEnter` and `OnMouseExit`) which allow you to do something unusual when the mouse starts moving over a button or leaves a button. These events obviously correspond to the time when the button flips from its inactive to active state (and vice versa). By utilising these events, you might (for example) change some aspect of the main window while the mouse is over a control. This might be a good way of allowing the user to 'preview' the effect of clicking a button without actually carrying out the action.

Ideas like this are fundamental to good user interface design. It's always a good idea to give users feedback on what will be the effect of carrying out some action. At the same time, you need to provide a way for a user to 'back off' from some action even after starting. For example, if you click and hold down the mouse on an Internet Explorer button, you'll see the button's appearance change to indicate that it's been pressed. However, if you then drag the mouse outside of the control (keeping the mouse button held down) you'll see the button's appearance change back to the inactive state. This is the control's way of saying "Yes, you've pressed me, but if you let go of the mouse now, we'll forget the whole thing – Ok?" I've tried to build the same user-friendliness into the button code presented here.

Reasons To Be Windowed

Last month, I referred to a button control that was included with the Issue 19 cover disk. Although this control claimed to be resource friendly, it actually allocated a `TForm` object behind the programmer's back, so to speak. When designing a Delphi control, it's very important to decide where to derive the control from and it makes a lot more sense to derive your new class from the right ancestor in the first place, rather than having to resort to horrible kludges like that. These days, with the advent of Windows 95 (which can cope with many more system resources than Windows 3.1), it really isn't a heinous crime to allocate a window handle! Some of the most innocuous VCL components do it. For example, did you know that `TTimer` allocates a window handle, even though it isn't (strictly speaking) a windowed control? Have a look at `TTimer.Create` (EXTCTRLS.PAS in the VCL source code) if you don't believe me!

The fact is, `TTimer` has to have a window handle because it uses API-level timer messages. These messages have to be sent to a window, hence the need for `TTimer` to allocate a small window for itself. This is sensible design: one extra window handle isn't going to break the bank. If `TTimer` didn't allocate a window, it would have to send the window messages to another windowed control, such as the parent form, and it would then have to sub-class the form's window procedure in order to receive the messages for itself. Sounds like a monumental kludge? You bet! You can see what I mean about choosing the right starting point for a control.

In exactly the same way, our Explorer button also needs a window handle. Any active control which doesn't actually have the input focus but needs to respond to mouse movement in it's 'air space' (so to speak!) must capture the mouse, and you can't capture the mouse unless you've got a window handle to send mouse messages to. Strictly speaking, you can use the VCL's `MouseCapture` property to capture mouse input even though your control isn't windowed: everything going back to `TControl` can use this run-time property. However, this is an illusion supported by clever coding in the VCL library – an illusion that breaks in a multi-threaded app because of the use of global variables.

Why the big deal about capturing the mouse? A button such as `TExplorerButton` needs to be able to capture the mouse so that all mouse-related messages are sent to it. If it didn't, there's the possibility that the mouse could be quickly 'snatched' away from the control, leaving the control in a highlighted state. Equally, once the mouse has been pressed down within the control, it's necessary to be able to track the movement of the mouse (even when the mouse moves outside of the control) so that the button knows how to display itself as the user drags it around the screen. I therefore chose to derive `TExplorerButton` from `TCustomControl`.

I've added a few other goodies to the button as well. A `Position` property is used to control the position of the glyph relative to the caption string. It can be to the left, right, above or below. In Figure 1, all the buttons have their `Position` property set to `bsTop`: glyph above. Another property, `Gap`, controls the distance between the glyph and the caption string. Three individual bitmap properties are used to supply the three bitmaps required for each of the buttons: active, inactive and disabled.

Frankly, this is one aspect of my control which I wasn't very happy with. Having to supply three different bitmaps for each control can be a bit tedious when you've got several controls in your toolbar. In principle, it should be possible to write code to take a single *active* bitmap and create monochrome and disabled versions of it. So, you should only need one `TBitmap` property instead of three. I looked into the possibility of doing this but decided that the amount of extra code involved was considerable and might be better covered in a future article dedicated to the subject. In the meantime, if you really want to do this now, and you're an MSDN member, then try scanning the MSDN CD-ROM for `BMUTIL` and `monochrome` and you'll find a lot of sample C code which can be converted into the equivalent Pascal.

Another thing I did was to expose the `Align` property. When you're populating a toolbar (such a `TPanel` component) with a number of these buttons, it's useful being able to set the `Align` property of each new button to `alLeft` so that it 'snaps' up against the previously added button.

Yet another property is `TransparentColor`, which allows you to set up a transparent colour for the three glyphs used by a particular button. As you'll probably appreciate, Windows only deals with rectangular bitmaps, which makes it rather problematic to paint a particular bitmap onto an arbitrarily coloured surface without leaving a rectangular border around the bitmap. Fortunately, the API contains some techniques which enable you to get around this, and Borland have thoughtfully provided this same functionality within the VCL through the use of the BrushCopy routine. Most button bitmaps, Borland's included, use clOlive as the transparent colour (probably because it's such a revolting colour that no-one in their right mind would want to use it in a bitmap for any other reason!), and I've therefore arranged that the TransparentColor property defaults to this value. You'll only need to change this property if you use any bitmaps with a different colour value for their background.

How It Works

Right then, let's get down to business and take a look at the code itself, which is given in Listing 1. There are two interesting aspects to this control: the way in which it paints itself and the manner in which it responds to mouse events. All the painting code is contained within (or called from) the TExplorerButton.Paint routine. The first job here is to fill in the background of the control using the current Color property. Next, the DrawFrame routine is called to optionally draw a 3D-frame around the control.

As noted earlier, we only get to draw a frame if the mouse is over the control (state equals bsActive) or if the mouse is currently pressed down over the control (bsDown). If the control is inactive, or the mouse has been pressed down and dragged outside of the control, then no frame is shown. This is for consistency with Internet Explorer. As an added convenience, the control detects if it's being used in design mode and if so draws a standard frame. If you don't do this, it's very hard to see where the control begins and ends while you're laying out a form! Depending on whether the control is active or depressed, the DrawFrame code then draws a raised or recessed frame.

Back in the main Paint routine, the code calculates bounding rectangles for the caption and glyph before passing the rectangles to the Layout routine. This in turn relocates the two bounding rectangles according to the current value of the Position property. One bit of sneakiness here is the way in which the Layout routine is called, conditionally swapping the two parameters before the call. By doing things this way, the Layout procedure only has to cater for two possibilities instead of four. The Layout code also takes care of offsetting both rectangles by one pixel if the button is recessed, so as to add to the 3D effect.

The only other thing worthy of note in the painting code is the way in which a disabled caption is drawn. This just uses the standard trick of drawing the same string twice (first shadowed and then highlighted) with a one pixel offset between the two. This gives the chiselled, 3D appearance of a disabled caption.

The mouse handling code (implemented in WMLButtonDown, WMLButtonMove and WMLButtonUp) looks simple but needs some careful thought. Most user interface controls which respond to the mouse can be thought of as simple 'state machines' and you need to consider all the possible states which are needed to correctly represent the behaviour of the control, plus the transitions that are needed to move from one state to another. There are also a number of non-obvious subtleties in the use of the VCL class library.

For example, controls which derive from TCustomControl will, by default, have the csCaptureMouse style bit set in the ControlStyle property. This style bit causes the control to automatically capture the mouse when the left-hand mouse button is clicked and not to 'let go' of the capture until the mouse button is released. Since this is pretty well what we want to happen, I left the csCaptureMouse bit set; you can turn these bits off in the control's constructor if you want. However, while developing the control, I forgot that calling the inherited WLMButtonUp routine will turn off the mouse capture, and this is definitely not what we want if the mouse is still over the control, for the reasons given above. That's why I have to set the MouseCapture property back to True inside my own WMLButtonUp handler. If I didn't do this, clicking the mouse button inside the control would effectively leave the button in what amounted to an invalid state for the control, and moving the mouse away from the control after clicking would leave the button in a highlighted state.

Maybe you think that these sort of deliberations sound a bit tortuous? Well, they probably are, but you need to face these sort of issues when designing well-behaved user interface controls. Some years ago, I spent 18 months of my life commuting from the Essex coast to a job in Uxbridge, Middlesex. Twice a day, I traversed the entire width of the London Underground network! I killed the time by reading and by doing those popular 'Logic Problems' books which you can buy from any newsagent. In retrospect, I feel the Logic Problems were excellent training for developing user interface controls!

Popup Menu Support

There's one other feature of our Explorer button that I haven't mentioned so far. If you take a look at some of the Microsoft buttons in Internet Explorer or Office97, you'll see that they have an associated popup menu. Clicking them causes a drop-down menu to appear immediately below the button. I wanted my button to have the same functionality.

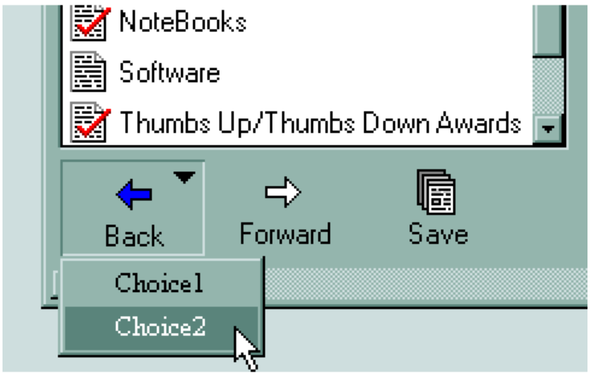
As it happens, this is pretty straightforward. As you'll know, the TCustomControl already provides access to the standard VCL PopupMenu property. We can use this property, in the normal way, to associate a popup menu with our button control. Inside the button code, we can look to see if the property has a non-Nil value and, if so, display the menu instead of behaving like a simple clickable button. However, a little further surgery is required before things happen the way we'd like.

For starters, PopupMenu is designed to respond to right-hand mouse clicks, whereas we want the menu to appear in response to a normal left-button click. In order to fix this, you'll see that I've added a new right-click handler which intercepts WM_RButtonDown messages. The new event handler simply sets the AutoPopup field in the associated popup menu, and then calls the inherited handler. It's the AutoPopup field which determines whether or not a menu will pop up in response to a right-hand mouse click. By setting this field to zero, we ensure that right mouse clicks are effectively ignored. You might be thinking that we could just as easily set the AutoPopup property (it's a published field) of the popup menu at design time. However, when designing a re-usable component, it's a good idea to make it as bullet-proof as possible. Never assume that the component user has set things up the way we'd like them to be!

It's also important to decide where we're going to set the popup menu's AutoPopup handler to False. Should we do it in the component's constructor? Should we do it whenever the assigned popup menu changes? By doing it in one place, in the right-click handler itself, we've automatically covered all the bases. This is because the inherited right-click handler is actually responsible for calling the popup menu's Popup method (see the VCL source) if the menu's AutoPopup flag is True. Thus, you can see why it's important to clear the flag before calling the inherited handler.

Having trampled all over the default action of the right-hand mouse button, we next need to turn our attention to left-hand mouse clicks. If you look at the left-click handler, you'll see that irrespective of whether or not there's a popup menu, I first set the button state to bsDown and draw the button in that state. This is for visual compatibility with Internet Explorer, which likewise draws buttons in the 'down' state while a popup menu is active.

Having done this, the code then checks to see if there's an associated popup menu and, if so, calculates the screen position at which the menu should appear. For a menu's Popup method, the X and Y values are always in screen coordinates (to allow you to draw a popup menu anywhere on the screen) which is why we need a call to ClientToScreen. My code is written so as to draw the popup menu immediately below the Explorer button. Going back to what I said earlier, the code makes as few assumptions as possible about the health (!) of the popup menu, and therefore it ensures that the Alignment property is set to paLeft immediately before the menu is displayed. Again, if you don't do this, then things won't look very nice if the menu happens to be set for a non-standard alignment.



➤ *Figure 2: This enlarged view of the Explorer button shows the small down-pointing arrow mark which is automatically displayed if the button has an attached popup menu*

Finally, the PopupComponent property of the menu is set to Self (this instance of TExplorerButton) and the menu's Popup method is called to actually display the menu (more on PopupComponent in a minute). Internally, the Popup method calls the API routine TrackPopupMenu and doesn't return until the menu has been dismissed. This is an important point because once we get back from the call, we know that it's safe to 'un-press' the button and return its state to bsInactive. Why inactive? Well, since the menu is positioned immediately below the button, it stands to reason that it we click the mouse button in the menu and the menu disappears, the mouse will no longer be located over the button, and therefore it's appropriate to go back to displaying the button in its inactive state.

Menu Initialisation Issues

As you'll no doubt appreciate, many applications modify the appearance of a menu according to certain criteria. For example, most programs disable the Paste menu item unless there's actually something on the Windows clipboard that can be pasted. It doesn't make sense to enable the Copy item unless there's a current selection which it makes sense to copy, and so forth. While it's possible to modify menu items as the various criteria change, it's much more convenient to simply set up the state of menu items in a particular drop-down menu immediately before the menu is displayed.

Delphi caters for this type of approach by providing an OnPopup event for popup menus. By writing an OnPopup event handler, you can get control immediately before a menu appears and use the opportunity to set up menu items as required. I decided not to publish an event handler for the Explorer button but rather to simply rely on the OnPopup functionality in the associated menu. In other words, if you want something special to happen immediately before your popup menu appears, just add an OnPopup handler to the menu and do your menu initialisation in there.

This raises the question of what to do if the same popup menu is used by more than one Explorer button. That's an unlikely scenario, but we need to cater for it by making sure that the menu's PopupComponent property is initialised before calling the Popup method. This identifies the component that's using the menu and allows us to do different things according to which Explorer button is currently using the menu. By examining the PopupComponent property inside the menu's OnPopup handler, you can determine which button is using it and therefore know which items to place into the menu.

As an extra bit of icing on the cake, I've added another feature to our Explorer button. You'll notice that Internet Explorer displays a small downward-pointing triangle in the top right-hand corner of a button to indicate that it has an associated drop-down menu. I've modified the `Paint` method of my button so that, in the same way, if the `PopupMenu` property of the button isn't `Nil`, a small drop-down menu mark will be displayed. I've hard-wired the size and position of the mark, but you might consider having the position of the mark change dynamically as the `Position` property is altered to change the layout of the control. If you wanted, you could also add another `TBitmap` property so that the user of the component could assign a custom mark. However, that's probably taking things a bit too far: as I've observed in the past, good component design needs to achieve an engineering compromise between complexity and flexibility!

The actual `DrawMenuGlyph` routine is the code responsible for drawing the triangle using the VCL's `Polygon` method. This technique is quick and easy to use, but be warned that it does have disadvantages. Like a number of other VCL routines, `Polygon` relies upon Delphi Pascal's open-array capability: the ability to pass an arbitrarily sized array to a routine. The implementation of open arrays in Delphi Pascal is cunning (see my previous articles on Delphi code generation), but it does generate a lot of code; if you're drawing a lot of very complex polygons using this approach, you'll get a substantial reduction in code size by switching to a series of simpler line drawing operations.

Finally, bear in mind that at design-time, assigning an Explorer button's `PopupMenu` property won't instantly cause the menu mark to appear or disappear. This is because there's no internal VCL message which is triggered by a change in this property. You'll remember that in last month's code, I exploited the `cm_EnabledChanged` message to cause a button to instantly redraw when it's `Enabled` property is changed, but there's no equivalent `cm_PopupMenuChanged` message. It's not a big problem, but if you reassign the `PopupMenu` property at run-time, you should call the control's `Refresh` method to ensure that the change is immediately apparent.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the author of *Instant Delphi Programming* published by Wrox Press. You can contact Dave as DaveJewell@msn.com, DSJewell@aol.com or DaveJewell@compuserve.com. Portions of this article first appeared in PC Pro magazine and are Copyright © 1995-1996, D S Jewell.

➤ *Listing 1 (continues overleaf)*

```
unit ExpBtn;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, Menus;
type
  TExpBtnState =
    (bsInactive, bsActive, bsDown, bsDownAndOut);
  TGlyphPosition = (bsTop, bsBottom, bsLeft, bsRight);
  TExplorerButton = class(TCustomControl)
  private
    fCaption: String;
    fInactive: TBitmap;
    fActive: TBitmap;
    fDisabled: TBitmap;
    fState: TExpBtnState;
    fMouseExit: TNotifyEvent;
    fMouseEnter: TNotifyEvent;
    fTransparentColor: TColor;
    fGlyphPosition: TGlyphPosition;
    procedure DrawFrame;
    procedure SetCaption(const Val: String);
    procedure SetInactiveGlyph(Val: TBitmap);
    procedure SetActiveGlyph(Val: TBitmap);
    procedure SetDisabledGlyph(Val: TBitmap);
    function CurrentGlyph: TBitmap;
    procedure SetTransparentColor(Val: TColor);
    procedure SetGlyphPosition(Val: TGlyphPosition);
    procedure Layout(var txtRect, bitRect: TRect);
  protected
    procedure Paint; override;
    procedure WMLButtonDown(var Message: TWMLButtonDown);
      message wm_LButtonDown;
    procedure WMRButtonDown(var Message: TWMRButtonDown);
      message wm_RButtonDown;
    procedure WMMouseMove(var Message: TWMMouseMove);
      message wm_MouseMove;
    procedure WMLButtonUp(var Message: TWMLButtonUp);
      message wm_LButtonUp;
    procedure CMEnabledChanged(var Message: TMessage);
      message cm_EnabledChanged;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Color;
    property Font;
    property Enabled;
    property ParentFont;
    property PopupMenu;
    property ShowHint;
    property ParentShowHint;
    property Visible;
    property OnClick;
    property Align;
    property OnDblClick;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property Caption: String read fCaption write SetCaption;
    property GlyphInactive: TBitmap
      read fInactive write SetInactiveGlyph;
    property GlyphActive: TBitmap
      read fActive write SetActiveGlyph;
    property GlyphDisabled: TBitmap
      read fDisabled write SetDisabledGlyph;
    property Position: TGlyphPosition read fGlyphPosition
      write SetGlyphPosition default bsTop;
    property TransparentColor: TColor read fTransparentColor
      write SetTransparentColor default clOlive;
    property OnMouseExit: TNotifyEvent
      read fMouseExit write fMouseExit;
    property OnMouseEnter: TNotifyEvent
      read fMouseEnter write fMouseEnter;
  end;
procedure Register;
implementation
constructor TExplorerButton.Create(AOwner: TComponent);
begin
  Inherited Create(AOwner);
  fInactive := TBitmap.Create;
  fActive := TBitmap.Create;
  fDisabled := TBitmap.Create;
  fState := bsInactive;
  fGlyphPosition := bsTop;
  fTransparentColor := clOlive;
  Width := 50; Height := 40;
end;
destructor TExplorerButton.Destroy;
begin
  fInactive.Free;
  fActive.Free;
  fDisabled.Free;
  Inherited Destroy;
end;
procedure TExplorerButton.CMEnabledChanged(
  var Message: TMessage);
begin
  Inherited;
  Invalidate;
end;
procedure TExplorerButton.SetInactiveGlyph(Val: TBitmap);
begin
  fInactive.Assign(Val);   { ** CONTINUED OVER THE PAGE --> }
```

```
{ ** LISTING 1 CONTINUED FROM PREVIOUS PAGE }
  Invalidate;
end;
procedure TExplorerButton.SetActiveGlyph(Val: TBitmap);
begin
  fActive.Assign(Val);
  Invalidate;
end;
procedure TExplorerButton.SetDisabledGlyph(Val: TBitmap);
begin
  fDisabled.Assign(Val);
  Invalidate;
end;
procedure TExplorerButton.SetCaption(const Val: String);
begin
  if fCaption <> Val then begin
    fCaption := Val;
    Invalidate;
  end;
end;
procedure TExplorerButton.SetTransparentColor(Val: TColor);
begin
  if fTransparentColor <> Val then begin
    fTransparentColor := Val;
    Invalidate;
  end;
end;
procedure TExplorerButton.SetGlyphPosition(
  Val: TGlyphPosition);
begin
  if fGlyphPosition <> Val then begin
    fGlyphPosition := Val;
    Invalidate;
  end;
end;
function TExplorerButton.CurrentGlyph: TBitmap;
begin
  { Default to inactive glyph - use others if present }
  Result := fInactive;
  if (fState in [bsActive, bsDown]) and
    (not fActive.Empty) then
    Result := fActive;
  if (not Enabled) and (not fDisabled.Empty) then
    Result := fDisabled;
end;
procedure TExplorerButton.DrawFrame;
var rClient: TRect;
    State: TExpBtnState;
    LT, BR: TColor;
begin
  State := fState;
  rClient := ClientRect;
  { If we're designing, draw component in 'Active' state }
  if csDesigning in ComponentState then State := bsActive;
  { Only Active and Down states have a border }
  if State in [bsDown, bsActive] then with Canvas do begin
    if State = bsActive then begin
      LT := clBtnHighlight; BR := clBtnShadow;
    end else begin
      LT := clBtnShadow; BR := clBtnHighlight;
    end;
    with rClient do begin
      Pen.Color := LT;
      MoveTo(Right - 1, 0); LineTo(0, 0);
      LineTo(0, Bottom - 1);
      Pen.Color := BR;
      MoveTo(1, Bottom - 1);
      LineTo(Right - 1, Bottom - 1);
      MoveTo(Right - 1, 1);
      LineTo(Right - 1, Bottom);
    end;
  end;
end;
procedure TExplorerButton.Layout(
  var txtRect, bitRect: TRect);
var hBit, vBit, hTxt, vTxt: Integer;
begin
  hBit := bitRect.Right - bitRect.Left;
  vBit := bitRect.Bottom - bitRect.Top;
  hTxt := txtRect.Right - txtRect.Left;
  vTxt := txtRect.Bottom - txtRect.Top;
  case fGlyphPosition of
    bsTop, bsBottom :
      begin
        bitRect.Left :=(Width - hBit) div 2;
        txtRect.Left := (Width - hTxt) div 2;
        bitRect.Top := (Height - (vBit + vTxt)) div 2;
        txtRect.Top := bitRect.Top + vBit;
      end;
    bsLeft, bsRight :
      begin
        bitRect.Top := (Height - vBit) div 2;
        txtRect.Top := (Height - vTxt) div 2;
        bitRect.Left := (Width - (hBit + hTxt)) div 2;
        txtRect.Left := bitRect.Left + hBit;
      end;
  end; { case }
  bitRect.Right := bitRect.Left + hBit;
  bitRect.Bottom := bitRect.Top + vBit;
  txtRect.Right := txtRect.Left + hTxt;
  txtRect.Bottom := txtRect.Top + vTxt;
  { If button down, draw text and glyph down and to right }
  if fState = bsDown then begin
    OffsetRect(bitRect, 1, 1);
    OffsetRect(txtRect, 1, 1);
  end;
end;
procedure TExplorerButton.Paint;
var
  x, y: Integer;
  Glyph: TBitmap;
  txtRect, bitRect, glyphRect: TRect;
  procedure DrawMenuGlyph(x, y: Integer; Color: TColor;
    Style: TBrushStyle);
  begin
    with Canvas do begin
      Pen.Color := Color;
      Brush.Color := clBlack;
      Brush.Style := Style;
      Canvas.Polygon([Point(x, y), Point(x + 8, y),
        Point(x + 4, y + 4)]);
    end;
  end;
begin
  with Canvas do begin
    Brush.Color := Color;   { Fill control background }
    Brush.Style := bsSolid;
    FillRect(ClientRect);
    DrawFrame; { Draw control frame - if applicable }
    { Figure out size of text and display bitmaps }
    Font := Self.Font;
    Glyph := CurrentGlyph;
    txtRect :=
      Rect(0, 0, TextWidth(Caption), TextHeight(Caption));
    bitRect := Rect(0, 0, Glyph.Width, Glyph.Height);
    glyphRect := bitRect;
    { Now calculate position of text and bitmap }
    if fGlyphPosition in [bsTop, bsLeft] then
      Layout(txtRect, bitRect)
    else
      Layout(bitRect, txtRect);
    { First, draw the caption }
    Brush.Style := bsClear;
    if Enabled then
      TextRect(txtRect, txtRect.left, txtRect.top, fCaption)
    else begin
      Font.Color := clBtnShadow;
      TextRect(txtRect, txtRect.left, txtRect.top, fCaption);
      OffsetRect(txtRect, 1, 1);
      Font.Color := clBtnHighlight;
      TextRect(txtRect, txtRect.left, txtRect.top, fCaption);
    end;
    { Draw the drop-down menu 'glyph' }
    if PopupMenu <> Nil then begin
      x := Width - 14; y := 4;
      if Enabled then begin
        if fState = bsDown then begin
          Inc(x);
          Inc(y);
        end;
        DrawMenuGlyph(x, y, clBlack, bsSolid);
      end else begin
        DrawMenuGlyph(x, y, clBtnShadow, bsClear);
        DrawMenuGlyph(x + 1, y + 1, clBtnHighlight, bsClear);
      end;
    end;
    { Finally, draw the glyph }
    Brush.Color := Color;
    BrushCopy(bitRect, Glyph, glyphRect, fTransparentColor);
  end;
end;
procedure TExplorerButton.WMRButtonDown(
  var Message: TWMRButtonDown);
begin
  { Disable AutoPopup before calling Inherited }
  if PopupMenu <> Nil then
    PopupMenu.AutoPopup := False;
  Inherited;
end;
procedure TExplorerButton.WMLButtonDown(
  var Message: TWMLButtonDown);
var pt: TPoint;
    InControl: Boolean;
begin
  Inherited;
  InControl := PtInRect(GetClientRect,
    Point(Message.XPos, Message.YPos));
  if InControl then begin
    MouseCapture := True;
    fState := bsDown;
    Invalidate;
    if PopupMenu <> Nil then begin
      pt :=
        Parent.ClientToScreen(Point(Left-1, Top+Height));
      PopupMenu.Alignment := paLeft;
      PopupMenu.PopupComponent := Self;
      PopupMenu.Popup(pt.x, pt.y);
      fState := bsInactive;
      MouseCapture := False;
      Invalidate;         { ** CONTINUED ON FACING PAGE --> }
```

```
{ ** LISTING 1 CONTINUED FROM FACING PAGE }
    end;
  end;
end;
procedure TExplorerButton.WMMouseMove(
  var Message: TWMMouseMove);
var InControl: Boolean;
begin
  Inherited;
  InControl := PtInRect(GetClientRect,
    Point(Message.XPos, Message.YPos));
  if(fState = bsDown) and (not InControl) then begin
    fState := bsDownAndOut;
    Invalidate;
  end;
  if (fState = bsDownAndOut) and InControl then begin
    fState := bsDown;
    Invalidate;
  end;
  case fState of
    bsInActive :
      if InControl then begin
        fState := bsActive;
        if Assigned(fMouseEnter) then
          fMouseEnter(Self);
        MouseCapture := True;
        Invalidate;
      end;
    bsActive :
      if not InControl then begin
        fState := bsInActive;
        if Assigned(fMouseExit) then
          fMouseExit(Self);
        MouseCapture := False;
        Invalidate;
      end;
  end;
end;
procedure TExplorerButton.WMLButtonUp(
  var Message: TWMLButtonUp);
var InControl: Boolean;
begin
  Inherited;
  InControl := PtInRect(GetClientRect,
    Point(Message.XPos, Message.YPos));
  if InControl then begin
    fState := bsActive;
    MouseCapture := True;
  end else begin
    fState := bsInactive;
    MouseCapture := False;
  end;
  Invalidate;
end;
procedure Register;
begin
  RegisterComponents('Pilgrim''s Progress',
    [TExplorerButton]);
end;
end.
```