Things You Can Do With Standard Controls: The TButton

by Brian Long

One of the most common complaints about the TButton is that it has no Color property, unlike similar controls in many 4GL packages. Another frequent whinge is that the Caption property is limited to one line of text. To resolve these two issues we can take advantage of the *owner draw* capabilities of the button. Delphi's TListBox, TComboBox and TOutline controls have properties which cater for reasonably easy custom-drawn versions (see the April 1997 issue for details on how to take of advantage of these facilities in TListBox and TComboBox). For some reason TButtons do not, despite Windows support for this capability.

The use of owner drawn controls tends to suggest the "owner" in the term is you, the programmer, because it is you who does the drawing. Technically, however, this is not the case. The term comes from Windows SDK programming. The idea in Windows is that the control's owner is responsible for drawing it. Before continuing, we will investigate these terms *owner* and *parent*. They have specific definitions in Windows and also in Delphi. On the Windows side of things, the terms relate to the Windows interface elements that Windows draws, as opposed to the components that Delphi uses to represent them and make them easier to work with, that is they relate to windows with window handles.

Owner Windows And Parent Windows

The relationship between owner window and owned windows defines which windows will be automatically got rid of by Windows. When an owner window is destroyed, all the windows owned by it are also destroyed. This is the principal aspect of the relationship, but additionally owned windows are always above their owners in the 3-D window layering or Z-order, and when an owner window is minimised, owned windows are hidden.

The relationship between parent window and child window defines the ability to draw on the screen. A child window can only draw itself in its parent window's client area.

When a window is created (with Delphi this is done in the CreateWnd method of a TWinControl) it can, generally speaking, be categorised as one of three types, as specified by flags in its window style (which can be set in the Delphi CreateParams method).

Firstly, an overlapped window, as specified by the ws_Overlapped style, is supposed to be used as the main application window in an API written program, and comes equipped with a caption bar, border and client area.

Secondly, a popup window, as specified by the ws_Popup style, is intended to be used as a message box, dialog or temporary window that can appear outside of the application window.

Lastly, a child window, as specified by the ws_Child style, is confined to the client area of its parent window. Unlike popup and overlapped windows, child windows cannot be owners but can be parents.

Delphi forms are normally overlapped windows apart from those with BorderStyle set to bsNone which are popup windows as are Delphi 2 and 3 forms that have been set to bsDialog. Table 1 shows which windows become parents and owners of other windows of the varying types.

The parent can be changed after window creation with a call to the SetParent API, but the owner cannot be changed. To find the owner of a window, call

```
GetWindow(Wnd, gw_Owner)
```

where Wnd is the window handle. This doesn't work for child windows as the parent and owner are one and the same and so Windows doesn't bother to store the owner information. To find the parent, you can use GetParent(Wnd) or

```
GetWindowWord(
  Wnd, gww_HWndParent)
```

which changes to

```
GetWindowLong(
  Wnd, gwl_HWndParent)
```

in Win32. The approaches for finding the parent and owner are summarised in Table 2.

Owner Components And Parent Components

Now let's look at Delphi components and consider the meaning that they impose on the terms parent and owner, or more correctly the Owner and Parent properties. Remember that Delphi components are memory objects whose task, in many cases, is to simplify the manipulation of Windows interface elements. When a Delphi component is created there is no Windows interface element, ie no window handle. That comes later, if at all.

Indeed as properties get modified, the window may get destroyed and recreated several times during the component's lifetime. Examples of this include setting the Sorted property of a TListBox and the BorderIcons property of a TForm. Windows doesn't support changing those attributes of a window after it has been created and so the VCL must destroy and recreate the window. At some point the component should

	Owner	Parent
ws_Overlapped	The desktop window	The desktop window
ws_Popup	Set by the WndParent parameter in a call to CreateWindow or CreateWindowEx; if this window is a child window, the owner is the first non-child parental ancestor of WndParent	The desktop window, unless one is explicitly set by a call to the SetParent API
ws_Child	The owner is implicitly the same as the parent	Set by the WndParent parameter to CreateWindow or CreateWindowEx

➤ *Table 1: Who can be an owner and a parent of a window*

	Owner	Parent
ws_Overlapped	GetWindow(Wnd, gw_Owner)	GetParent(Wnd) returns 0, ie HWnd_Desktop; can also use GetWindowLong(Wnd, gwl_HWndParent)
ws_Popup	GetWindow(Wnd, gw_Owner)	GetParent(Wnd) confusingly returns the owner, so use GetWindowLong(Wnd, gwl_HWndParent); if there is no parent, this will return the owner
ws_Child	GetWindow(Wnd, gw_Owner) returns 0 so you can use GetParent(Wnd) since the parent is the same as the owner	GetParent(Wnd)

➤ *Table 2: Finding owner and parent windows*

get destroyed. If there is a valid window handle at this point, the represented window gets destroyed also.

In the context of components, the Owner property can refer to any other component, not just one that represents a window (that is, not just TWinControls and descendants). The point of a component's owner is to unburden the programmer from the requirement of calling a component's destructor or its Free method. If you, the programmer, don't destroy a component, the component's owner will do the job during its own destruction. Therefore the component's memory will be de-allocated, and any other appropriate tidying up will be done. This is quite separate from the idea of a window's owner.

It is only visual components (that is, TControls and their descendants) that have a Parent property, not all components. Similar to window parents, a TControl's Parent property dictates the screen space on which the control can draw. However a TControl doesn't necessarily have a window handle and so does not necessarily represent a Windows interface element. For example a TSpeedButton has no window handle (and consequently it can't take the focus): it is manually drawn by code inside the component class.

It is a TControl descendent, TWinControl, that adds support for Windows interface elements. The component that becomes the Parent must be a TWinControl, or a component inherited from one. If a TWinControl has a Parent, then that Parent will be a TWinControl-based component that represents its parent window.

Changing Parent And Owner

A component's owner can be changed by calling the owner's RemoveComponent method and then any other component's InsertComponent method: you cannot assign a component to the Owner property as it is read-only. The parent can be changed by assigning a TWinControl to the Parent property.

Owner Drawn Components

Let's get back to how an owner drawn control is supposed to work. The idea is for the owner window to implement the drawing of the owned window. That means, if you have a button that needs custom drawing, the button's owner does the drawing. The button's owner window will be the same as its parent window (since a button is created with a ws_Child style), which will be the form or panel or group or notebook page etc that it has as its Parent property.

When customising components' functionality we tend to write event handlers. These tend not to live in the Parent component, but in the form. As it happens, the form is set as the Owner of all components placed on the form at design time. Therefore, when we set out to do custom drawing of buttons, listboxes, combo boxes etc, which is supposed to be done by the owner window, we will actually do it in the Owner component. So in Delphi, owner drawn components should be called Owner drawn components. It is probably inconsequential, as you are no doubt thinking!

The way owner drawn controls work is by changing their window style during window creation. A special value indicating a desire for owner drawing is used. At some point, Windows sends the owner window a wm_MeasureItem message, and the owner fills in the parameters telling Windows the coordinates. In the case of listboxes and combo boxes, this would be the size of one item in the listbox. In the case of variable sized owner draw listboxes and combo boxes, this would be sent for each item that needed to be drawn on the screen. The Windows documentation states that wm_MeasureItem needn't

be processed for owner draw buttons.

When Windows wants the control to be drawn it sends the owner window a `wm_DrawItem` message, and the owner is required to do the drawing. The plan is for the control to have custom drawing, but the control's code need not be modified: its owner is modified instead. This is a bit like the Delphi event model: when you want to customise a component, you write an event handler inside another class, not in the component. However, Windows messages and Delphi events are not the same thing and so we should see how the one gets mapped to the other.

Components that support owner drawing have `OnDrawItem` and `OnMeasureItem` events that get triggered at appropriate points. The events are not in the owner window component. The event handlers might end up there but the events are in the component itself. This would imply that the owner window component passes information back to the owner drawn component to allow the events to fire. How is this managed?

Notification Messages

These messages that get sent by Windows to a control's owner are generically termed notification messages. Delphi helps keep all of the functionality used to represent and manipulate a window under the auspices of one component by introducing a new set of parallel messages called component notifications. These messages are manufactured by the VCL and the message numbers start at `cn_Base` (`$2000`), rather above all the system messages. The idea is that the Delphi component representing a control's owner/parent might receive notification messages for the control. To allow the component to completely implement all the required functionality for the control, the parent sends the message straight back to the control. To avoid any confusion though, the message is first turned into a component notification.

Table 3 shows the two normal Windows messages that have so far been described and the equivalent component notifications. You can see that to turn the message into a component notification, the parent adds `cn_Base` to it and then sends it off to the relevant control. This is done in the `DoControlMsg` subroutine in the `Controls` unit, which is called by message handlers for all the notification messages received by any `TWinControl`.

Message	Value
wm_MeasureItem	$2C
wm_DrawItem	$2B
cn_MeasureItem	cn_Base + wm_MeasureItem = $202C
cn_DrawItem	cn_Base + wm_DrawItem = $202B

➤ *Table 3: Windows notification and component notification messages*

32-bit app button in Windows 95

16-bit app button in Windows 95

➤ *Figure 1*

The net effect of this is to allow the control component to implement its own owner draw functionality by implementing message handlers for the component notification messages (although as mentioned, buttons don't need to react to `cn_MeasureItem`).

Note that the `TListBox`, `TComboBox` and `TOutline` components that already support owner drawing make a point of still working as normal when told to be owner drawn by their properties, until the appropriate event handler is put in place. To make the components feel quite complete, when told to be owner drawn, the components deal with the drawing themselves. It is only when an event handler becomes associated with the owner drawing event that they relinquish their responsibilities and let the event handlers do the work.

Owner Drawn Buttons

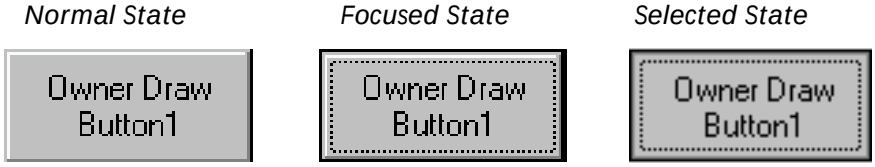
The `TButton` will only draw itself grey by default, or whatever colour the Control Panel specifies a button face is. It has no way of drawing itself in any more imaginative colours. This is because the `TButton` is a simple representative of a Windows button control, and Windows draws buttons using colours defined in the Control Panel. In terms of Delphi colour constants, the colours used are shown, admittedly not to scale, in Figure 1. Note that in Windows 95 a normal button on a 16-bit Delphi app and on a 32-bit Delphi app are drawn slightly differently.

In truth the point made above regarding the monotonous colour scheme is not entirely true. In Win32 applications, buttons can be drawn in a different colour by responding to the Delphi component notification message `cn_CtlColorBtn`, which is sent by a button's parent when it receives a `wm_CtlColorBtn` message. The `cn_CtlColorBtn` message can be trapped in a `TButton` descendant component, or the `wm_CtlColorBtn` message can be trapped in the form, if the button is placed on a form.

As mentioned, `TButtons` don't surface the inherent owner draw support in Windows, so we'll have to build a new component. However the `TBitBtn` class, which draws glyphs in various places, is an owner draw button, so can we rip most of the code from there? Well, that's not a bad idea. Indeed the `Buttons` unit (home of the `TBitBtn`)

has a very handy routine called DrawButtonFace that can draw the various lines and rectangles that make up an assimilation of both a Windows 3.1x type button and a Windows 95 type button. This could be very useful inside our new TButton derivative, since it should be able to draw itself until the user supplies a drawing event handler (to be consistent with other owner drawn components).

➤ *Figure 2*



➤ *Listing 1*

```
TDrawButtonEvent = procedure(Button: TOwnerDrawButton;
  Canvas: TCanvas; Rect: TRect; State: TOwnerDrawState;
  Default: Boolean) of object;
TOwnerDrawButton = class(TButton)
private
  FCanvas: TCanvas;
  FColor: TColor;
  FDefault,
  FOwnerDraw: Boolean;
  FOnDrawButton: TDrawButtonEvent;
protected
  procedure CreateParams(var Params: TCreateParams); override;
  procedure CNDrawItem(var Msg: TWMDrawItem); message cn_DrawItem;
  procedure WMLButtonDblClk(var Msg: TWMLButtonDblClk);
    message wm_LButtonDblClk;
  procedure DrawButton(Canvas: TCanvas; const Rect: TRect;
    State: TOwnerDrawState);
  procedure SetButtonStyle(ADefault: Boolean); override;
  procedure SetColor(Value: TColor);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure SetOwnerDraw(Value: Boolean);
published
  property Color: TColor read FColor write SetColor default clBtnFace;
  property OwnerDraw: Boolean read FOwnerDraw write SetOwnerDraw;
  property OnDrawButton: TDrawButtonEvent
    read FOnDrawButton write FOnDrawButton;
end;
```

➤ *Listing 2*

```
procedure TOwnerDrawButton.SetOwnerDraw(Value: Boolean);
begin
  if FOwnerDraw <> Value then begin
    FOwnerDraw := Value;
    RecreateWnd;
  end;
end;
procedure TOwnerDrawButton.CreateParams(var Params: TCreateParams);
const
  Styles: array[Boolean] of Longint = (0, bs_OwnerDraw);
begin
  inherited CreateParams(Params);
  with Params do
    Style := Style or Styles[FOwnerDraw];
end;
```

➤ *Listing 3*

```
procedure TOwnerDrawButton.CNDrawItem(var Msg: TWMDrawItem);
var State: TOwnerDrawState;
begin
  with Msg.DrawItemStruct^ do begin
    FCanvas.Handle := HDC;
    FCanvas.Font := Font;
    FCanvas.Brush := Brush;
    FCanvas.Brush.Color := FColor;
    { The Value typecast to Word is for cross-platform }
    { compatibility. In Win32 ItemState is a Longint. The }
    { typecast gives the low word, which is what we want }
    State := TOwnerDrawState(WordRec(Word(ItemState)).Lo);
    DrawButton(FCanvas, rcItem, State)
  end;
end;
```

Additionally, there is a Win32 API called DrawFrameControl that can draw many controls for us including buttons. The trouble with both these is that they will only draw a button in normal button colours. One of the points of this control is to cater for different color requests. So we can get some ideas from the TBitBtn, but not too much code.

The resultant class for an owner-draw button is in ODBUTTON.PAS and is shown in Listing 1.

What do the methods do? Well, the constructor sets up the default colour and builds a canvas object for use during owner-drawing and the destructor frees it. So nothing exciting there.

When the OwnerDraw property is set, the SetOwnerDraw method is called. If the new value is different from the old value it causes the Window button to be destroyed and rebuilt from scratch by the RecreateWnd call (see the article in the April 1997 issue on listbox customisations for more details on this window reconstruction process). The code in CreateParams ensures the window is built with the correct style for owner drawing (see Listing 2).

If the button is in an owner draw state, then the cn_DrawItem message handler will be invoked when necessary (see Listing 3). That sets the canvas up with an appropriate device context handle (supplied by a message parameter) and sets up the button's font and colour. Finally it calls the DrawButton method which either draws the button, or calls an event handler to do the job if one exists.

DrawButton is quite a long routine, as the job of drawing a button is quite involved. As shown in Figure 1 there are various lines in different colours. When the button is pushed in, things change, and when the button has focus additional bits need drawing: the thicker black border and the dashed rectangle around the caption (see Figure 2).

The code that draws all these masterpieces looks something like Listing 4, but as you can see most of it has been chopped out for

brevity. The important point is that the drawing only occurs if no event handler has been assigned.

Three other methods need mentioning before we leave this topic (shown in Listing 5). Firstly, when the new `Color` property is modified, `SetColor` is called. This stores the new colour value and makes sure the button redraws itself.

Secondly. with a normal `TButton`, when you go tabbing round a form and land on a button it gets a thicker border indicating it is selected. This is done in `TButton.SetButtonStyle` by toggling the window style of the button between `bs_PushButton` and `bs_DefPushButton`. This behaviour needs to be changed in the owner drawn button since the `bs_OwnerDraw` style will not co-exist with any style other than the default `bs_PushButton`. If we let this happen, the first time `SetButtonStyle` changes it to `bs_DefPushButton`, the button will no longer be owner drawn. To fix this, the new version changes a data field which is checked in the owner drawing code.

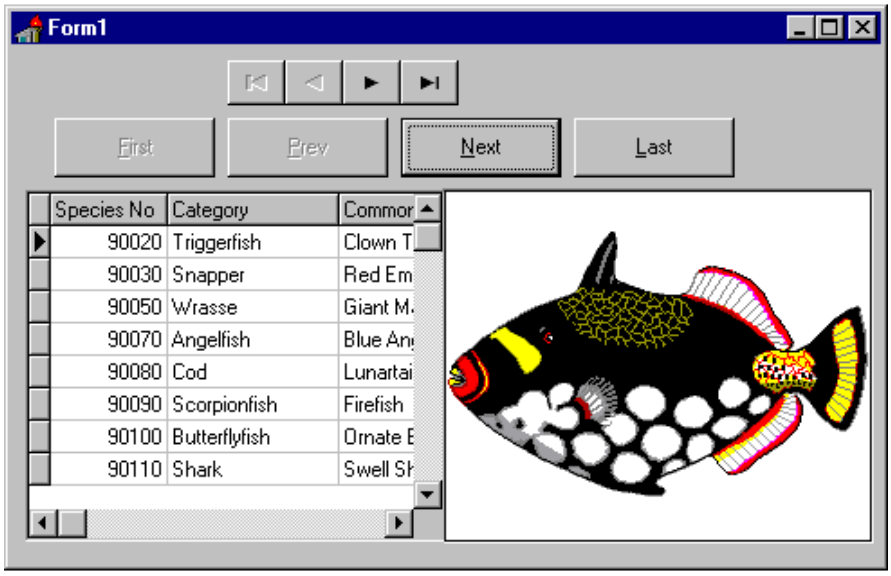
The last method is a `wm_LButtonDblClk` message handler. Without this, if you very quickly do several clicks on the button, it will only depress every alternate click, unlike normal buttons. This code ensures that on the alternate clicks, which are taken as double-clicks, the button acts just like it does on a normal single click.

The project ODBTNTST.DPR on this month's disk has an owner drawn button on the form, along with some other controls. At design time, the button draws itself with a multi-line caption and an operable `Color` property. At run-time, the button's `OnDrawButton` event handler is used to draw the button on the form. It also writes a number on the form's caption bar to indicate how many times it has been called. This event handler can be modified to draw the button in any fashion you like, but is not necessarily required if you just want a coloured button.

Auto-Repeating Buttons

The `TDBNavigator` component consists of a number of speed buttons.

➤ *Figure 3*



➤ *Listing 4*

```
procedure TOwnerDrawButton.DrawButton(Canvas: TCanvas;
  const Rect: TRect; State: TOwnerDrawState);
var
  ... { full code on this month's disk }
begin
  if Assigned(FOnDrawButton) then
    FOnDrawButton(Self, Canvas, Rect, State, FDefault)
  else
    ... { Draws button as in a 16-bit app under Windows 95 }
    ... { full code on this month's disk }
end;
```

➤ *Listing 5*

```
procedure TOwnerDrawButton.SetButtonStyle(ADefault: Boolean);
begin
  { Normally, when focus is received, TButton changes
    its style to give the thicker black border. But when
    owner drawn, you are not allowed other styles }
  if FOwnerDraw then begin
    FDefault := ADefault;
    Refresh;
  end else
    inherited SetButtonStyle(ADefault);
end;

procedure TOwnerDrawButton.SetColor(Value: TColor);
begin
  if FColor <> Value then begin
    FColor := Value;
    if not FOwnerDraw then
      OwnerDraw := True
    else
      InValidate;
  end;
end;

procedure TOwnerDrawButton.WMLButtonDblClk(var Msg: TWMLButtonDblClk);
begin
  Perform(wm_LButtonDown, Msg.Keys, Longint(Msg.Pos));
end;
```

➤ *Listing 6*

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  BtnFirst.Enabled := not Table1.BOF;
  BtnPrior.Enabled := not Table1.BOF;
  BtnNext.Enabled := not Table1.EOF;
  BtnLast.Enabled := not Table1.EOF;
end;
```

If you hold down certain of these buttons (Prior and Next), they auto-repeat, ie they repetitively trigger the action they are associated with. You get much the same effect with keys on your keyboard. If you are typing in an edit control and hold down a key, it auto-repeats at a certain rate. Windows doesn't offer this functionality in buttons: the navigator had to manufacture it. If we want the same functionality in our own buttons we can manufacture it in the same way.

The project BTNRPT.DPR on this month's disk does just this. It has several normal buttons on the form, and uses various event handlers and a TTimer to achieve the effect. Additionally, the project shows how to get your own navigation buttons to disable when appropriate like the navigator's (see Figure 3). That particular task is done in the OnDataChange event of the data source connected to the relevant dataset (see Listing 6).

There are only two buttons on the form that need to auto-repeat, the Next and Prev buttons, and so it's those that we will focus on. As well as a normal OnClick handler, they also have OnMouseDown, OnMouseUp and OnMouseMove event handlers (see Listing 7). The logic goes like this. If the mouse is pushed in, then start a timer ticking and keep it going until the mouse is released. If, while the mouse is pushed in, the mouse moves off the button, the timer is paused. If the mouse moves back on the button, restart the timer.

This mimics the behaviour of the navigator. Indeed if you push any button and then move the mouse off it before releasing, the button pops back up to indicate that if you release the mouse button, the button won't get a click event.

Since a timer is required, one can simply be placed on the form or created upon demand. This project takes the former option: it's easier, after all.

To make this functionality more self-contained, it also comes supplied as a component in BTNREPT.PAS, which is used in a second version of the above project called BTNRPT2.DPR. This project is a duplicate of the first but uses the new TRepeatBtn components for the Next and Prev buttons instead of TButtons.

➤ *Listing 7*

```
procedure TForm1.NavBtnMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Timer1.Interval := InitRepeatPause;
  Timer1.Enabled := True;
end;

procedure TForm1.NavBtnMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
var
  Pt: TPoint;
begin
  { Even if mouse goes off button, mouse move events
    will still go to the button because it was created
    with the csCaptureMouse control style }
  Pt.X := X;
  Pt.Y := Y;
  if Timer1.Enabled then
    with (Sender as TButton) do
      Timer1.Enabled := PtInRect(Rect(0, 0, Width, Height), Pt)
end;

procedure TForm1.NavBtnMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Timer1.Enabled := False;
end;

procedure TForm1.TimerExpired(Sender: TObject);
var
  ActiveCtrl: TControl;
begin
  Timer1.Interval := RepeatPause;
  try
    ActiveCtrl := ActiveControl;
    (ActiveCtrl as TButton).Click;
    { If button has been disabled as a result of what }
    { happens in its OnClick method, shutdown timer }
    Timer1.Enabled := ActiveCtrl.Enabled;
  except
    Timer1.Enabled := False;
    raise;
  end;
end;
```

Conclusion

So there you have it. Now I've shown you the way, I'm sure that you will have endless fun designing and building your own owner-drawn buttons!

Brian Long is a UK-based freelance Delphi and C++ Builder consultant and trainer. He is available for bookings and can be contacted by email at blong@compuserve.com