Under Construction: Custom Events

by Bob Swart

In the last four columns we have examined the process of writing our own visual and non-visual components, components based on DLL engines, and adding custom bitmaps and component help files. This time, we'll focus on adding custom events and event handlers to our components.

Signallers And Handlers

Events consist of two parts: an event signaller and the event handler. The signaller must make sure that the component somehow gets a message of some sort to indicate that some condition has become true and that the event is now born. The event handler, on the other hand, starts to work only after the event itself is generated and responds to it by doing some processing of itself.

Event signallers are typically based on virtual (or dynamic) methods of the class itself (like the general `Click` method) or Windows messages, such as notification messages. Event handlers are typically placed in event properties, such as the `OnClick` or `OnChange` event handler property. If event handlers are published, the user of the component can enter some event handling code that is to be executed when the event is fired.

Event Handlers

Event Handlers are methods of type `Object`. This means that they can be assigned to class methods, and not to ordinary procedures or functions (the first parameter must be a `Self` type of thing). Consider the type `TNotifyEvent` for the most general of event handlers:

```
TNotifyEvent =
  procedure(Sender: TObject)
  of object;
```

The `TNotifyEvent` type is the type for events that have only the sender as parameter. These events simply notify the component that a specific event occurred at a specific `TObject` (the sender). For example, `OnClick`, which is type `TNotifyEvent`, notifies the control that a click event occurred on the control `Sender`. If the parameter `Sender` were omitted as well we'd only know that a specific event had occurred, but we'd not know to which control it had occurred. Generally, we do want to know for which control the event just occurred, so we can act on the control (or on data in the control).

As mentioned before, event handlers are placed in event properties, and they appear on a separate page in the Object Inspector (to distinguish them from the 'normal' properties). The basis on which the Object Inspector decides to split these two kinds of properties is the `procedure/function of Object` part of the declaration. The `of Object` part is needed as we get the error message *'cannot publish property'* if we omit it, as you can see in BOBEVENT.PAS in Listing 1. So, the event page only shows true event handlers, ie methods that are `of object`, and not just function pointers.

Event Signallers

Event signallers are needed to signal to an event handler that a certain event has occurred, so the event handler can perform its action. Event signallers are typically based on virtual (or dynamic) methods of the class itself (like the general `Click` method) or Windows messages, such as notification messages.

In order to give an example of an event signaller, we need an event handler to talk about. Let's examine the standard class `TListBox` from Delphi's VCL. This listbox has several event handlers: `OnClick`, `OnDblClick`, `OnDragDrop`, etc. I couldn't find an `OnChange` event handler, however, and this might be one I'd want to use more from day to day than an event handler like `OnKeyUp` or `OnMeasureItem`. Why

➤ *Listing 1*

```
unit BobEvent;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TEventNoObject = procedure;
  TEventOfObject = procedure of Object;
  TEventComponent = class(TComponent)
  private
    { Private declarations }
    FEventNo: TEventNoObject;
    FEventOf: TEventOfObject;
  published
    { Published declarations }
    property OnEventNoObject: TEventNoObject read FEventNo write FEventNo;
    { error: this property cannot be published }
    property OnEventOfObject: TEventOfObject read FEventOf write FEventOf;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Dr.Bob', [TEventComponent])
end;
end.
```

was OnChange omitted? I don't know, but it's a good thing Borland keeps providing me with these kinds of examples (thanks guys!).

OnChange

Before we just define our OnChange event, let's examine the VCL source code, to find out if OnChange is already present (and just not published). That might have been the easiest approach. Unfortunately (or fortunately, depending whether you're the reader or writer of this article), this is not the case, so we indeed have to create our own OnChange event handler and signaller for TListBox!

The OnChange event handler needs information about who (which listbox class) it is that has just received an OnChange message, so we need to have the so-called Sender. I would also like to know the previously selected item from the listbox (so I can compare it to the current ItemIndex, and see if we're currently walking up or down – might come in handy one day). If we need these two parameters, then our TChangeEvent type will look like the following definition (note the of Object part):

```
type
  TChangeEvent =
   procedure(Sender: TObject;
     PrevItemIndex: Integer)
   of Object;
```

TChangeEvent is just the type for the OnChange property; the property itself needs to be defined in a derived class of TListBox, including the private field FOnChange. But let's first concentrate on the event signaller.

Changed

Now, what can we use as an OnChange event signaller? In order to answer that question, we have to think of the different possible causes of this event. When will the position of a listbox change? Well, if the user just performed a keyboard or mouse action, then the position could have changed, for instance. And those keyboard and mouse actions already generate an OnClick event, so perhaps we can chain to that. All we need to do to find out whether or not the position has really changed is to keep track of the old position and compare it with the new position as soon as we get the Click notification message. Since we wanted to keep track of the previous position anyway (the PrevItemIndex), this shouldn't be a problem. We just have to add a private field FPrevItemIndex of type Integer to our derived TListBox, initialise it to -1, and set it to the current position after each change (ie at the end of each OnChange event).

Reading the VCL source code again, it turns out that the OnClick event handler itself is fired by the dynamic method Click of TListBox (which is defined at the TControl level, by the way). If we just override the Click method, we can fire the OnChange event (as well as performing the default behaviour, which will be firing the OnClick event).

TListBob

I've named the new listbox class TListBob for now and the new class definition can be seen in Listing 2.

Testing, 1, 2, 3...

If we add this component to our component palette, drop it on a form, and try to connect some code to the OnChange event handler, then everything seems to work as expected. If we click with the mouse on another item in the listbox, or we use the keyboard keys to select a new item, the OnChange event is fired whenever the position changes. Wonderful, it seems.

'Seems', is right, because we're not done yet (it would be a short column this month if we were, wouldn't it?). There is another situation that can cause the current position of the listbox to change and we didn't think of it before. Borland provides us with all those handy properties, like the

➤ *Listing 2*

```
unit ListBob;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TChangeEvent = procedure(Sender: TObject; PrevItemIndex: Integer) of
     Object;
  TListBob = class(TListBox)
  private
    { Private declarations }
    FPrevItemIndex: Integer;
    FOnChange: TChangeEvent;
  protected
    { Protected declarations }
    procedure Click; override;
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
  published
    { Published declarations }
    property OnChange: TChangeEvent read FOnChange write FOnChange;
  end {TListBob};
procedure Register;
implementation
constructor TListBob.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FPrevItemIndex := -1 { default: no selection }
end {Create};
procedure TListBob.Click;
begin
  inherited Click;
  if FPrevItemIndex <> ItemIndex then
    if Assigned(FOnChange) then FOnChange(Self, FPrevItemIndex);
  FPrevItemIndex := ItemIndex
end {Click};
procedure Register;
begin
  RegisterComponents('Dr.Bob', [TListBob]);
end;
end.
```

`ItemIndex` which indicates the index of the current selected item, or the current position in the listbox. Unfortunately, for `TListBob` this property is not read-only, but can be used to programmatically set the selected position:

```
with ListBob1 do
  ItemIndex := ItemIndex + 1;
```

Since this statement will surely not cause the `Click` method to fire, we won't get the `Change` signal and hence the `OnChange` event handler will not be executed. We were close, but no cigar!

WM_SetCurSel

So, we need another way to ensure that we're notified when the current selection of the listbox is set to another value. Fortunately, Windows itself provides us with this notification mechanism, as a `WM_SETCURSEL` message is fired whenever someone tries to set the current selection of a listbox. If we just listen for this message, we can fire the `OnChange` event right after the message itself, see Listing 3.

Note that in this case we don't even have to check if the `ItemIndex` has indeed changed from before to after the call, since Windows will make sure for us the message is only sent on a `WM_SETCURSEL` (and that one will only be fired if it is actually necessary, not when the desired position was actually already selected – it seems we actually benefit from some Windows optimisations for a change!). We do have to save the new position (`ItemIndex`) to our own saved `FPrevItemIndex` here as well, of course.

Two Event Signallers

Time for our second attempt at `TListBob`, one with two event signallers (`Click` and `WMSetCurSel`) and one event handler (`FOnChange`): see Listing 4.

Now we're in back business. The position in the listbox can be changed by the keyboard, mouse or programmatically, and we still manage to keep track of the changes. To illustrate this, I've written a little test that keeps the text of a `TEdit` up-to-date whenever a change in the `TListBob` occurs (and hence the `OnChange` event is fired): see Figure 1.

Example

The relevant parts of code are shown in Listing 5. First we have the `ListBox1Change` to actually update the `TEdit.Text` and then the four speedbutton event handlers to programmatically update the selection in the listbox.

Conclusion

We've seen that adding custom events to our Delphi components actually consists of two steps: adding event handlers (properties and types) and event signallers to fire the handlers. The `TListBox` component provided us with a nice

➤ *Listing 3*

```
procedure TListBob.WMSetCurSel(var Message: TMessage);
{ if someone assigns a new value to ItemIndex }
begin
  DefaultHandler(Message);
  if Assigned(FOnChange) then FOnChange(Self, FPrevItemIndex);
  FPrevItemIndex := ItemIndex
end {WmSetCurSel};
```

➤ *Listing 4*

```
unit ListBob;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TChangeEvent = procedure(Sender: TObject; PrevItemIndex: Integer) of
    Object;
  TListBob = class(TListBox)
  private
    { Private declarations }
    FPrevItemIndex: Integer;
    FOnChange: TChangeEvent;
  protected
    { Protected declarations }
    procedure Click; override;
    procedure WMSetCurSel(var Message: TMessage); message LB_SETCURSEL;
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
  published
    { Published declarations }
    property OnChange: TChangeEvent read FOnChange write FOnChange;
  end {TListBob};
procedure Register;
implementation
constructor TListBob.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FPrevItemIndex := -1 { default: no selection }
end {Create};
procedure TListBob.Click;
begin
  inherited Click;
  if FPrevItemIndex <> ItemIndex then
    if Assigned(FOnChange) then FOnChange(Self, FPrevItemIndex);
  FPrevItemIndex := ItemIndex
end {Click};
procedure TListBob.WMSetCurSel(var Message: TMessage);
{ if someone assigns a new value to ItemIndex }
begin
  DefaultHandler(Message);
  if Assigned(FOnChange) then FOnChange(Self, FPrevItemIndex);
  FPrevItemIndex := ItemIndex
end {WmSetCurSel};
procedure Register;
begin
  RegisterComponents('Dr.Bob', [TListBob]);
end;
end.
```

example to extend with the OnChange event.

Full code for the TListBob component and this example program is provided on the free disk this month, of course. Actually, the TListBob component on disk has another extra feature: automatic tabstop settings, so check it out.

Also on the disk is Version 2.0 of TBUUCode, the UUCode component for which we developed on-line component help last issue. It appears that I left two subtle bugs inside the 1.0 code, so to make it up to you I've included the new source code on disk – and guess what have been added to this component? An OnError and OnSuccess event handler, thanks to this month's column, of course...

Next Time

From now on, we're going to expand this column to include the creation of Delphi *experts* (IDE experts, that is, not self-proclaimed experts!) as well as components, since this is an area which many developers are interested in and (as you have found from my article in Issue 3) is not so difficult as you might think!

In the next issue, we'll be focusing on so-called property editors, like the one for glyphs on TBitBtn components, and will find out what property editors have in common with experts and components, and how we can write our own custom property editors for our own components and even for existing components!

Stay tuned, and make sure you've always got a backup of your COMPLIB.DCL in a safe place!

Bob Swart (you can email him at 100434.2072@compuserve.com) is a professional 16- and 32-bit software developer using Borland Pascal, C++ and Delphi. In his spare time, he likes to watch video tapes of Star Trek Voyager with his 1.75 year old son Erik Mark Pascal.

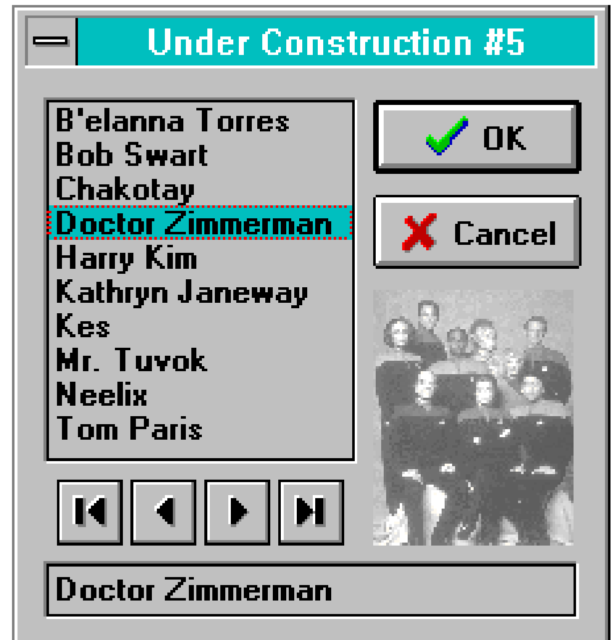

Figure 1: TListBob in action

Listing 5

```
procedure TForm1.ListBob1Change(Sender: TObject; PrevItemIndex: Integer);
begin
  Edit1.Text := Listbob1.Items[Listbob1.ItemIndex]
end;
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  Listbob1.ItemIndex := 0
end;
procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  Listbob1.ItemIndex := Pred(Listbob1.Items.Count)
end;
procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  if Listbob1.ItemIndex > 0 then
    Listbob1.ItemIndex := Listbob1.ItemIndex - 1
end;
procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  if Listbob1.ItemIndex < Pred(Listbob1.Items.Count) then
    Listbob1.ItemIndex := listbob1.ItemIndex + 1
end;
```

Would you like to tell Delphi users in 37 COUNTRIES all about your great new Delphi add-on?

OF COURSE YOU WOULD!

This space could be making sales for YOU if your advert was here!

We can produce your artwork and it needn't cost a fortune either: prices start from just £60 for a single insertion, with good series discounts. To find out more contact us using the details on Page 66.