Getting started with Lazarus: Basic components

Michaël Van Canneyt

January 1, 2007

Abstract

In this article, various standard components of the Lazarus Component Library (LCL) are examined in more detail, and various important properties are discussed. All of the controls described in this contribution are located on the first tab of the component palette in the Lazarus IDE.

1 Controls

Lazarus comes standard with a lot of components pre-installed, more than 100. More than half of these are actual visual components, `TControl` descendents. These visual components come in 3 flavours: regular controls, and a subset of the same controls in a data-aware version and a RTTI-aware version. The data-aware version of the controls can display and edit fields in a dataset. RTTI controls can be coupled to a published property of any class, and allow displaying and editing the property value.

All visual controls descend from TControl. As such they share a lot of properties such as position and size, and anchor properties, together with a `OnResize` event, all of which have been treated in previous contributions.

Besides the position and size-related properties, the following properties are shared by all controls:

Color the background color of the control.

Cursor the shape of the mouse cursor when the mouse is over the control.

Enabled if set to `False`, the control will not receive any events, and will not receive focus. Depending on the kind of control, in general it will be displayed in a different color, indicating it is not available.

Font If the control displays some text, the font used to display the text.

Hint A hint that can be shown in a small popup window as the mouse moves over the control. This is also known as a tooltip.

ShowHint if set to `True`, the hint will popup if the mouse is over the control. If `False`, no hint will be shown.

ParentColor If set to `True`, the `Color` property of the control is always equal to the `Color` property of the control's parent.

ParentFont Similar to `ParentColor`: if set to `True`, then the `Font` property of the

ParentShowhint If set to `True` (the default), the value of `Showhint` is always the same as the `ShowHint` property of the control's parent. This means that setting the `Showhint` property of a form to `True`, will enable hints for all controls on the form. control's parent is used. This can e.g. be used to quickly reset a font.

They also share the following events:

OnClick Triggered when the user clicks the control with the left mouse button.

OnDblClick Triggered when the user double-clicks the control with the left mouse button.

OnMouseEnter Triggered when the mouse crosses the boundaries of the control and starts moving over the control.

OnMouseLeave Triggered when the mouse crosses again the boundaries of the control and starts moving away from the control.

OnMouseMove Triggered when the mouse moves over the control.

OnMouseDown Triggered when a mouse button is pressed while the mouse is over the control.

OnMouseUp Triggered when a pressed mouse button is released while the mouse is over the control.

These events can be used to respond to mouse movement and mouse events. The `OnMouseDown` and `OnMouseUp` events are more fine-grained events than the `OnClick` and `OnDblClick` events: a control may receive a `OnMouseDown` event, but not a `OnMouseUp` event: in that case no `OnClick` event will be generated. This can be the case when the user presses the mouse button on one control, moves the mouse to another control, and only then releases the mouse button.

These properties and events will be demonstrated in the following section.

2 The label control

The label control (the `TLabel` class) can be used to show a text in a single font and color. It's biggest use is as an accompagnying text with other controls such as edits, memos and so on. It introduces the following properties:

Alignment is the horizontal alignment: it determines whether the text in the label is aligned left, right or is centered. The effect of this property is not really visible when `Autosize` is `True`.

Autosize determines whether the label resizes itself so it is big enough to show the caption. If it's set to `False`, and the text is too big, the text is truncated.

Caption The text to display in the label. If a character in the text is preceded by an ampersand (&), then it will appear underlined, and will function as an accelerator character: pressing the 'Alt' and the character key simultaneously will shift focus to the control referred to in `FocusControl`. This behaviour can be disabled with the `ShowAccelChar` property. To actually display an ampersand, 2 ampersands must be entered (&&).

FocusControl The control which will receive focus when the accelerator key is pressed. This must be a `TWinControl` descendent, i.e. a control which can actually receive focus.

Layout is the vertical alignment, it determines whether the text is aligned to the top, bottom or is centered. Again, if the property `Autosize` is `true`, setting `Layout` has no visible effect.

ShowAccelChar Determines whether the ampersand (&) character in the `Caption` denotes an accelerator key.

Transparent If `True`, the background is not colored prior to drawing the label, i.e., the `Color` property is ignored.

WordWrap if enabled, a text which is too long for the label's width will be wrapped, i.e., text which does not fit on a single line will be broken into several lines. This property has no effect if `Autosize` is `True`. Note that hard linefeeds in the caption will still be respected if `WordWrap` is `True`.

To demonstrate all these properties, a small demo program is coded. It's a simple form, with a series of labels, a button and 2 checkboxes. The button (`BClose`) simply closes the form, the checkboxes control some of the properties.

There are 5 labels:

LHelp contains a simple instruction text. It's a multiline label, no wordwrap.

LURL contains a simple text. The various mouse event handlers are implemented to display some status information and create a hypertext-kind of effect.

LPos displays the position of the mouse as it moves over the `LURL` label. It has the `Autosize` property set to `True`.

LButton displays a message when a mouse button is pressed on the `LURL` label. It has the `Autosize` property set to `False`. The effect of this can be seen e.g., by keeping ctrl-shift pressed when clicking the `LURL` label.

LBClose is linked with it's `FocusControl` property to the `BClose` button. The `B` character is the accelerator character: pressing `ALT-B` will shift focus to the button (but will not generate a click).

All labels have a `Hint` property, and have the `ParentShowHint` set to true.

The first of the 2 checkboxes (`CBEnableHints`) controls the `ShowHint` property of the form: if it's checked all labels will show a hint. Note that the `ParentShowHint` of the checkbox itself is set to `False` and that the `ShowHint` is set to `True`: as a result the checkbox will always show a hint when the mouse is over it.

The second of the 2 checkboxes (`CBEnableAccelerator`) controls the `ShowAccelChar` property of the `LBClose` label. It can be used to test what happens if the property is set to `False`. The `Hint` property of this checkbox is empty: no matter what the value of the `ShowHint` property of the form is, no hint will be shown.

The various mouse event handlers of the `LURL` label are quite simple:

```
procedure TMainForm.LURLMouseEnter(Sender: TObject);
begin
  LURL.Font.Color:=clBlue;
end;

procedure TMainForm.LURLMouseLeave(Sender: TObject);
begin
  LURL.Font.Color:=clBlack;
```

```
  LPos.Caption:='Mouse pos:';
  LButton.Caption:='Mouse Button:';
end;
```

The `OnMouseEnter` event handler simply switches the color of the label text to blue. The `OnMouseLeave` event resets the text color to black, and resets the caption of the 2 status labels.

The mouse events that are triggered when the mouse is over the `LURL` label are equally simple:

```
procedure TMainForm.LURLMouseMove(Sender: TObject;
                                  Shift: TShiftState;
                                  X, Y: Integer);
Var
  S,T : String;
begin
  S:=Format('Mouse pos: (%d,%d)',[X,Y]);
  T:=ShiftStateToString(Shift);
  If (T<>'') then
    S:=S+'Shift: '+T;
  LPos.Caption:=S;
end;

procedure TMainForm.LURLMouseDown(Sender: TObject;
                                  Button: TMouseButton;
                                  Shift: TShiftState;
                                  X, Y: Integer);

Var
  S,T : String;

begin
  S:=MouseButtonToString(Button);
  S:=format('%s Mouse Button: down at (%d,%d)',[S,X,Y]);
  T:=ShiftStateToString(Shift);
  If (T<>'') then
    S:=S+'Shift: '+T;
  LButton.Caption:=S;
end;

procedure TMainForm.LURLMouseUp(Sender: TObject;
                                Button: TMouseButton;
                                Shift: TShiftState;
                                X, Y: Integer);
Var
  S,T : String;

begin
  S:=MouseButtonToString(Button);
  S:=format('%s Mouse Button: up at (%d,%d)',[S,X,Y]);
  T:=ShiftStateToString(Shift);
  If (T<>'') then
    S:=S+'Shift: '+T;
  LButton.Caption:=S;
```

```
end;
```

The `OnMouseMove` handler simply displays the current position. The `OnMouseDown` and `OnMouseUp` handlers show that the event took place, and where it took place.

The `Button` parameter for the last 2 event handlers has one of the following values: `mbLeft`, `mbRight`, `mbMiddle`, denoting the left, right and middle mouse button, respectively. The `MouseButtonToString` function converts this value to a readable text.

The `Shift` parameter to these events denotes the state of the special keys when the event takes place. It's a set which can have the following values included:

ssShift the Shift key is depressed.

ssAlt the Alt key is depressed.

ssCtrl the Ctrl key is depressed.

ssLeft left mouse button is down.

ssRight right mouse button is down.

ssMiddle middle mouse button is down.

ssAltGr the Alt-gr key is depressed (right alt key).

ssCaps Caps lock key.

ssNum Num lock key.

The `ShiftStateToString` function converts the Shift set to a string.

The event handlers for the button and the 2 checkboxes is simple, and is just given for completeness:

```
procedure TMainForm.BCloseClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.CBEnableAccelleratorChange(Sender: TObject);
begin
  LBClose.ShowAccelChar:=CBEnableAccelerator.Checked;
end;

procedure TMainForm.CBEnableHintsChange(Sender: TObject);
begin
  ShowHint:=CBEnableHints.Checked;
end;
```

The resulting application is shown in figure 1 on page 6.

3 The Panel control

The panel control (`TPanel`) is similar to a Label control in that it displays a text. However, it can do much more than that: it's a windowed control, so other controls can be dropped on it (no controls can be dropped on a label). The fact that other controls can be dropped on

Figure 1: The labels demo program

it can be very useful for grouping components, or to get certain layout effects (as demonstrated in a previous contribution). Additionally, the panel can draw a border around itself and can be used as a tab stop.

`TPanel` shares many properties with `TLabel`, but has some additional properties that control the 2 bevels (which make up the panel's border) around the panel:

BevelInner the appearance of the inner bevel.

BevelOuter the appearance of the outer bevel.

BevelWidth the width of the bevels.

The `BevelInner` and `BevelOuter` properties can have the values `bvNone`, `bvLowered`, `bvRaised` and `bvSpace`, with obvious meanings.

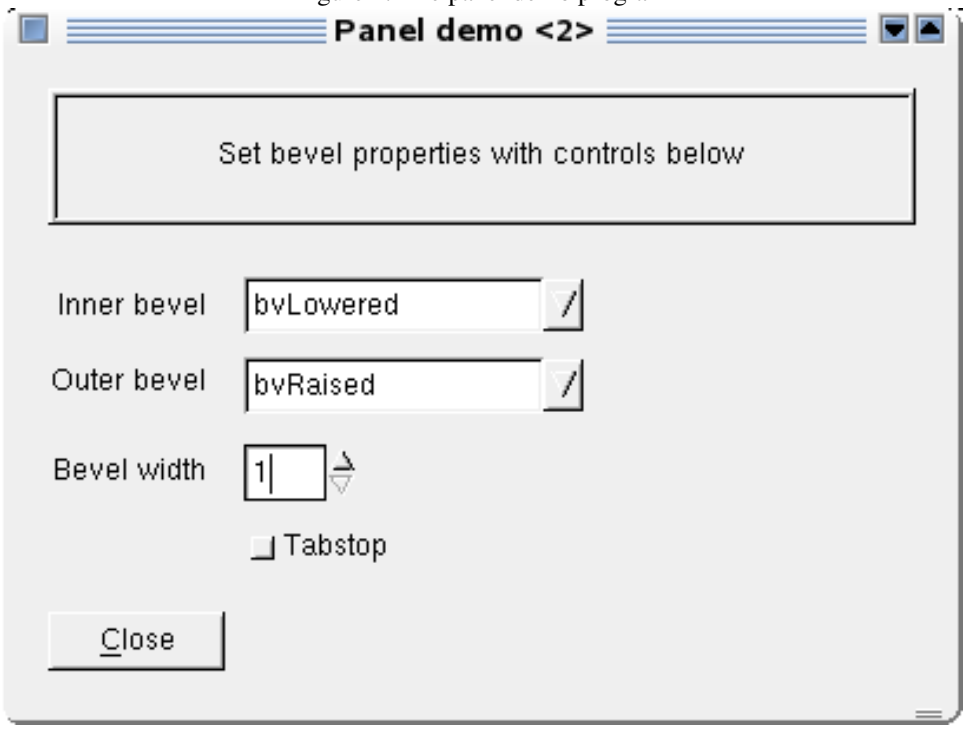
Other than that it has a bevel the `TPanel` is a windowed control: this means that other controls can be dropped on it. It also means that it can receive focus, and participates in the tab order of the form. This behaviour is controlled by the `TabStop` property: if it is set to `True`, then the panel can be focused when the user presses the `Tab` key. By default, it's set to false.

The paneldemo program shown in figure 2 on page 7 demonstrates the above properties. There is no code in the program: some RTTI controls are used to control the panel's properties directly.

4 The bevel control

The bevel control (`TBevel`) is visually related to the panel control: it shows a border, or part of a border, but without any caption. The 'inside' of the bevel is empty, and no controls

Figure 2: The panel demo program



can be dropped on it. It's just a control which draws a frame along its borders: it can be put around other controls to group them visually (just like a panel).

Unlike a panel, `TBevel` is not a windowed control. Therefore it does not participate in the tab order of the form: the controls inside it are part of the tab order of the control on which the bevel is located (usually, the form)

There are 2 properties which control the bevel's appearance:

Shape this can be a box (`bsBox`), frame (`bsFrame`) or any of the 4 edges of the bounding rectangle (`bsTopLine` etc.).

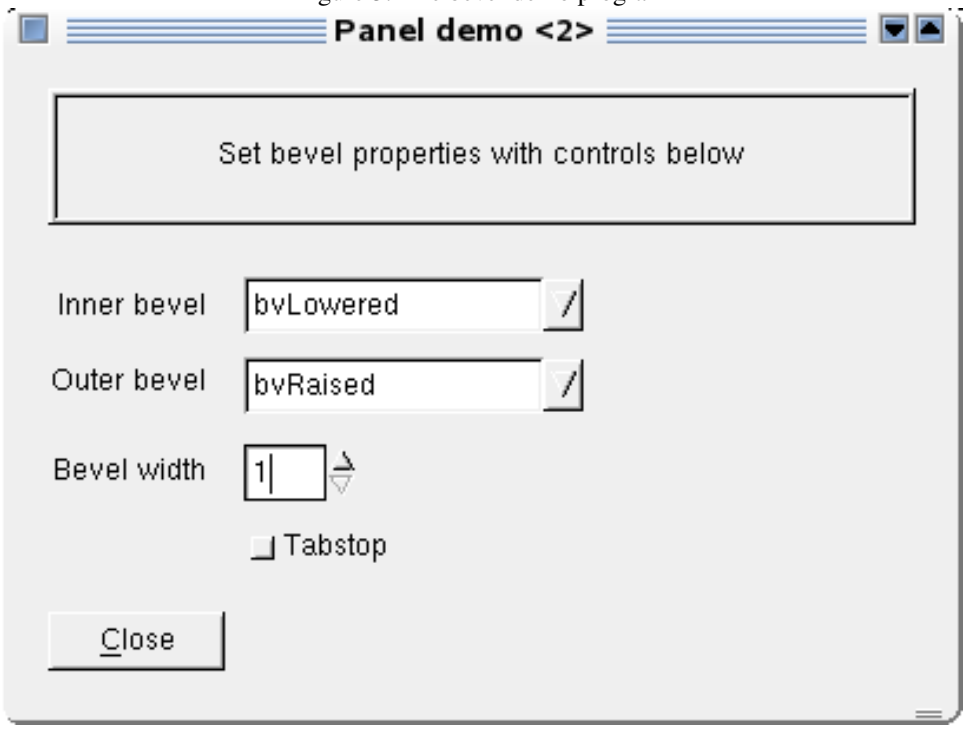
Style this is `bsLowered` or `bsRaised` and determines whether the lines look engraved or enbossed.

They are demonstrated in the beveldemo demonstration program (figure 3 on page 8). The bevel is anchored to the right edge of the form: Resizing the form will move the bevel. The button - which is initially centered in the bevel will remain on it's place, because it is not dropped on the bevel. To show the difference, a panel which looks similar to the bevel is also dropped on the form, with a button on top. When resizing the form, the difference in behaviour will immediatly be obvious.

5 The groupbox control

The groupbox control (`TGroupBox`) shares some properties with a bevel and a panel. It draws a frame, and displays a caption in the top-left corner of the frame. like the panel, it can contain controls, and can receive focus. It's commonly used to group controls, and provies a title for the group. The title can have an accelerator character, which can be used to put focus on the groupbox.

Figure 3: The bevel demo program



The groupbox control has no additional distinguishing properties. Setting the caption and using the `ChildSizing` property (discussed earlier) should be all the customization that is needed.

6 The button control

The button control (`TButton`) and it's companion BitButton (`TBitButton`) are simple button controls: They display a text (and in the case of BitButton an image), and the user can click on them. The text is specified in the `Caption` property, and can contain an accelerator character which can be used to activate the button.

To make construction of standardized dialog forms easier, some extra properties exist:

Cancel The button acts as the 'Cancel' button on the form: pressing the 'Escape' key will activate the button, usually to close the dialog without saving changes.

Default The button acts as the default button of the form (usually the 'OK' button): it will be activated if the user presses the 'Enter' or 'Return' key on the keyboard.

ModalResult If this is set to any other value than `mrNone`, then clicking the button will set the `ModalResult` property of the form on which the button is located, and this will automatically close the form if it is shown modal. Note that the `OnClick` handler is still executed: it can be used to perform saving of data if needed. As soon as the handler has been executed, the form is closed.

Combining these properties makes it easy to construct standardized dialogs which close themselves when the user presses Escape or Return. The former should discard any changes made in the dialog, the latter should save all changes.

The BitButton control has some more properties to control it's layout:

Glyph The image to show on the button if the `Kind` property is `bkCustom`.

Kind By default this is 'bkCustom', meaning that a custom image and caption can be specified. If this is set to any of the other values (bkOK, bkCancel, bkHelp, bkYes, bkNo, bkClose, bkAbort, bkRetry, bkIgnore, bkAll, bkNoToAll, bkYesToAll), default values for the `Glyph`, `Caption` and `ModalResult` properties are filled in.

layout Determines the location of the glyph relative to the caption. It can have one of the following values: `blGlyphBottom`, `blGlyphLeft`, `blGlyphRight` and `blGlyphTop`, with obvious meanings.

Margin This is the amount of pixels between the button border and the image on the button. Setting it to -1 will center the text and image on the button.

NumGlyphs This is the number of images in the `Glyph` bitmap. The first will be used as the standard image, the second as the image when the button is disabled.

Spacing This is the amount of pixels between the caption and the image on the button.

Both buttons are windowed controls, which means they can receive focus. If a button is specified in the `FocusControl` property of a label, then pressing the accelerator character of the label will focus the button, but will not cause the `OnClick` event to be fired. The accelerator character in the button's caption, in contrast, will trigger the `OnClick` event.

All these properties are demonstrated in the buttondemo program. The main form demonstrates the 'Default' and 'Cancel' properties: pressing the 'Enter' key or 'Escape' key will activate them: Note that they need not be connected with a certain value of the `ModalResult` property.

The `ModalResult` property is demonstrated in a separate window, where the `ModalResult` property can be set with a radiogroup - choosing one of the values will set the `ModalResult` property of the button to the chosen value. The program contains almost no code except the opening of the auxiliary windows:

```
procedure TMainForm.BBitButtonsClick(Sender: TObject);
begin
  With TBitButtonDemoForm.Create(Self) do
    begin
    Show;
    end;
end;

procedure TMainForm.BModalResultClick(Sender: TObject);

Var
  mr : TModalresult;
  msg : string;

begin
  With TModalForm.Create(Self) do
    Try
      mr:=ShowModal;
      msg:=RGModalResult.Items[RGModalResult.ItemIndex];
    finally
      Free;
```

```
    end;
  ShowMessage(Format('Result : %d (%s)',[mr,msg]));
end;
```

Which message to show when the default or cancel button are pressed, is determined using the `Tag` property of the buttons: This is the primary use for the `Tag` property: in case multiple controls are connected to the same event handler (as in the demo program), it can be used to determine which control has actually triggered the event using a `case` statement. Note that the `Tag` property is introduced in `TComponent`, and since the `Sender` parameter is of type `TObject`, a typecast is needed:

```
procedure TMainForm.BCancelClick(Sender: TObject);

Var
  Msg : String;

begin
  Case TComponent(Sender).Tag of
    1 : Msg:='Default button pressed';
    2 : Msg:='Cancel button pressed';
  else
    Msg:='Error: Unknown sender !'
  end;
  ShowMessage(Msg);
end;
```

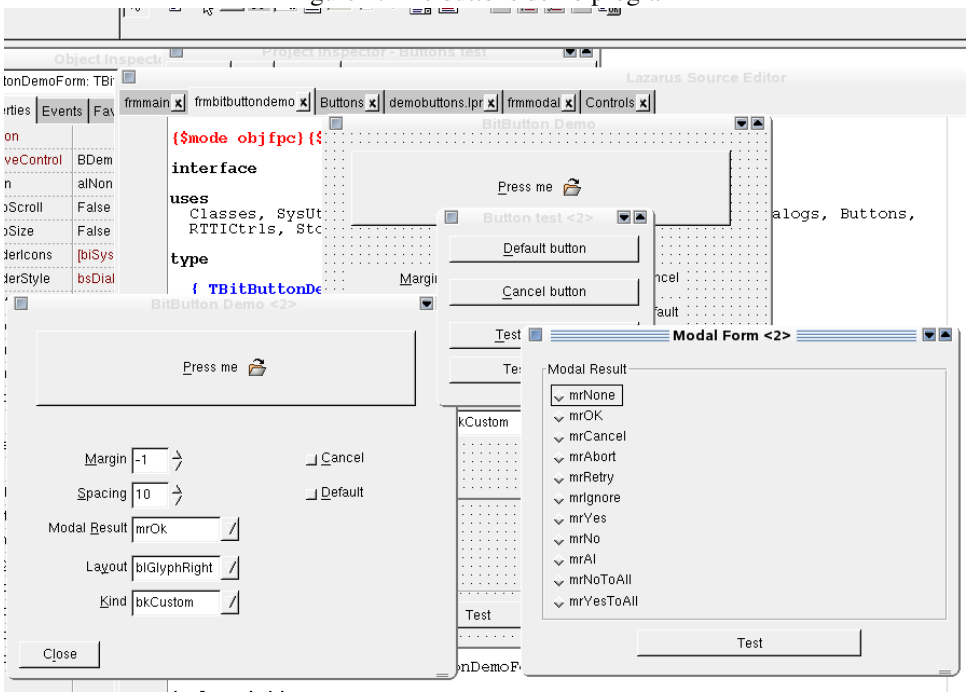
Obviously, in case there are not too much controls, the `Sender` property can be compared to the actual controls as in the following code:

```
procedure TMainForm.BCancelClick(Sender: TObject);

Var
  Msg : String;

begin
  If (Sender=BDefault) then
    Msg:='Default button pressed'
  else if (Sender=BCancel) then
    Msg:='Cancel button pressed'
  else
    Msg:='Error: Unknown sender !'
  end;
  ShowMessage(Msg);
end;
```

7 The CheckBox control

The checkbox control (`TCheckBox`) is similar to the Button control in the sense that it is clickable: The `OnClick` event is triggered when the check mark is toggled by clicking it or hitting the space bar if the checkbox has focus. The fact that both the `TCheckBox` and `TButton` classes descend from the `TButtonControl` class, reflects this similarity.

Figure 4: The buttons demo program



Checkboxes are used whenever a use needs to enable or disable some options. Multiple checkboxes can be used to choose multiple options.

Except from the `Caption` property and the `OnClick` event handler which it has common with the `TButton` control, the checkbox has the following properties which determine it's behaviour and appearance:

Autosize as for the `TLabel` control, `AutoSize` determines whether the checkbox adjusts it's width to accomodate for the size of the caption (`True`), or whether it keeps a fixed size (`False`).

AllowGrayed determines whether the checkbox can appear in the grayed state.

Checked determines whether a checkbox appears in the control.

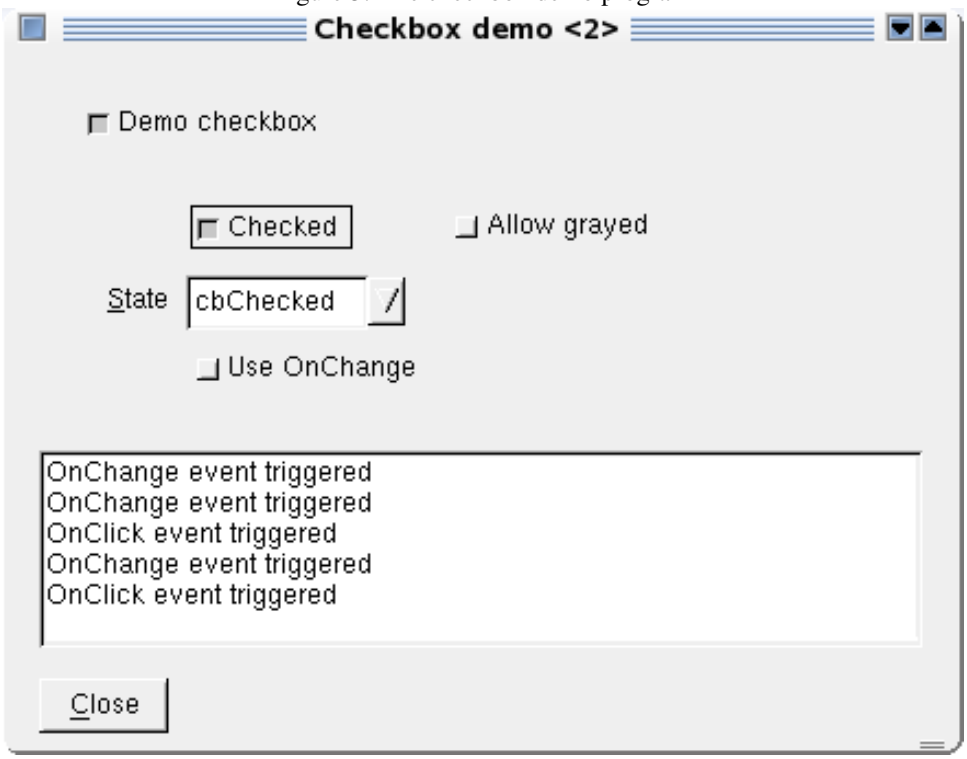
State this is one of `cbUnchecked`, `cbChecked` or `cbGrayed`

UseOnChange Determines what happens if the `Checked` property is changed in code. Normally, when the checked property changes (due to a user action or in code), an `OnClick` event is triggered and a `OnChange` event as well. If the `UseOnChange` property is set to `True`, only a `OnChange` event is triggered when the `Checked` property is changed in code (this is the Delphi compatible behaviour).

All these properties are demonstrated in the checkboxdemo program (figure 5 on page 12). The `OnClick` and `OnChange` handlers of the `CBDemo` demo checkbox write a line to determine what happens when the checked property changes:

```
procedure TMainForm.CBDemoClick(Sender: TObject);
begin
  MLog.Lines.Add('OnClick event triggered');
end;
```

Figure 5: The checkbox demo program




```
procedure TMainForm.CBDemoChange(Sender: TObject);
begin
  MLog.Lines.Add('OnChange event triggered');
end;
```

Note that setting the `State` property of the checkbox does not trigger any event at all.

8 The ToggleBox control

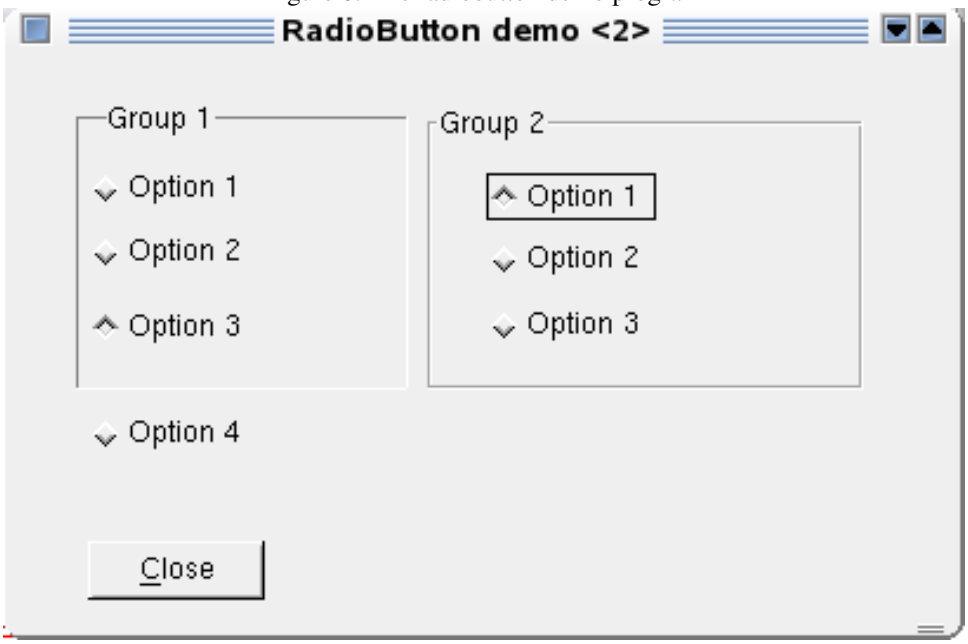
The ToggleBox control (`TToggleBox`) has the look of a regular button control, but behaves like a checkbox: it remains 'checked' (depressed) when the user clicks it. To 'uncheck' it, it must be clicked again. It has the same properties and behaviour (including the `OnChange` and `OnClick` handlers) as a checkbox control. A demonstration program similar to the one for the checkbox is provided.

9 The RadioButton control

The RadioButton control (`TRadioButton`) is similar to a checbox. It has the same properties as the Checkbox. In difference with a checkbox, only one radiobutton can be checked at a time: checking one radiobutton will clear all other checkboxes in the same group. Therefore a radiobutton can only be used to let a user choose between mutually exclusive options. If multiple options can be chosen at the same time, a checkbox must be used.

Radiobuttons belong to the same group if they have the same parent control. This parent

Figure 6: The radiobutton demo program



control can be anything: a form, a panel, a groupbox. This is demonstrated in a small application (figure 6 on page 13), which contains 4 radiobuttons that are located on the form, and 3 located on a groupbox. Out of the 4 radiobuttons on the form, 3 are surrounded by a bevel. Despite this, they still form a group with the fourth radiobutton, since they share the same parent: the form. The radiobuttons on the groupbox form a separate group, their parent control is the groupbox: even though the groupbox' parent is the form, they still form a separate group. Only the direct parent is taken into account.

10 The RadioGroup control

The radiogroup control (`TRadioGroup`) is a combined control: it's simply a groupbox which contains a nymber of radiobuttons. The captions of the radiobuttons are determined by the `Items` property. Which radiobutton in the group is selected is determined by the `ItemIndex` property.

There are 2 properties that determine the layout of the radiogroup:

Autofill determines whether the options are evenly distributed along the height of the control: if not, the options are just placed one after the other, with no space between them. If `Autofill` is enabled (the default) then enough whitespace is put between the options so the complete groupbox is filled.

ColumnLayout determines the direction how the columns are filled with options: horizontally or vertically.

Columns the number of columns in the groupbox.

They are all demonstrated in the demogroupbox application.

11 The CheckGroup control

The Checkgroup control (`TCheckgroup`) is similar to the radiogroup. Instead of radiobuttons, checkboxes are used, which means that the user can select multiple options. In this case, the `ItemIndex` property no longer makes sense. Instead the `Checked` array property can be used to check which of the items were checked. The `CheckEnabled` property can be used to disable certain options.

The demogroupbox application can be changed to demonstrate the `checked` property: an additional button and a memo (`MLog`) are added to the form. In the buttons `OnClick` handler, the following code is executed:

```
procedure TMainForm.BCountClick(Sender: TObject);

Var
  I,Total: Integer;

begin
  MLog.Lines.Clear;
  Total:=0;
  For I:=0 to CGDemo.Items.Count-1 do
    If CGDemo.Checked[i] then
      begin
      MLog.Lines.Add(CGDemo.Items[i]+' is checked');
      Inc(Total);
      end;
  MLog.Lines.Add(Format('%d items checked.',[Total]));
end;
```

This verifies which items are checked by the user. The checked items are displayed in the memo, followed by a total. An additional button is placed on the form to disable (or enable) the second item:

```
procedure TMainForm.CBEnableItem2Click(Sender: TObject);
begin
  With CGDemo do
    If Items.Count>1 then
      CheckEnabled[1]:=Not CheckEnabled[1];
end;
```

Note that enabling or disabling an item has no influence on the fact whether it can be checked or not. This can be easily verified with the counting mechanism. The result can be seen in figure 7 on page 15

12 Conclusion

In this article, a lot of the standard controls on the Lazarus component palette have been examined. Each has it's particular properties, and all of them have been discussed and demonstrated in demo programs. The controls discussed here don't do much, but make up for a large part of the controls that are used daily. In the next contribution, more complex controls will be examined.

Figure 7: The checkgroup demo program