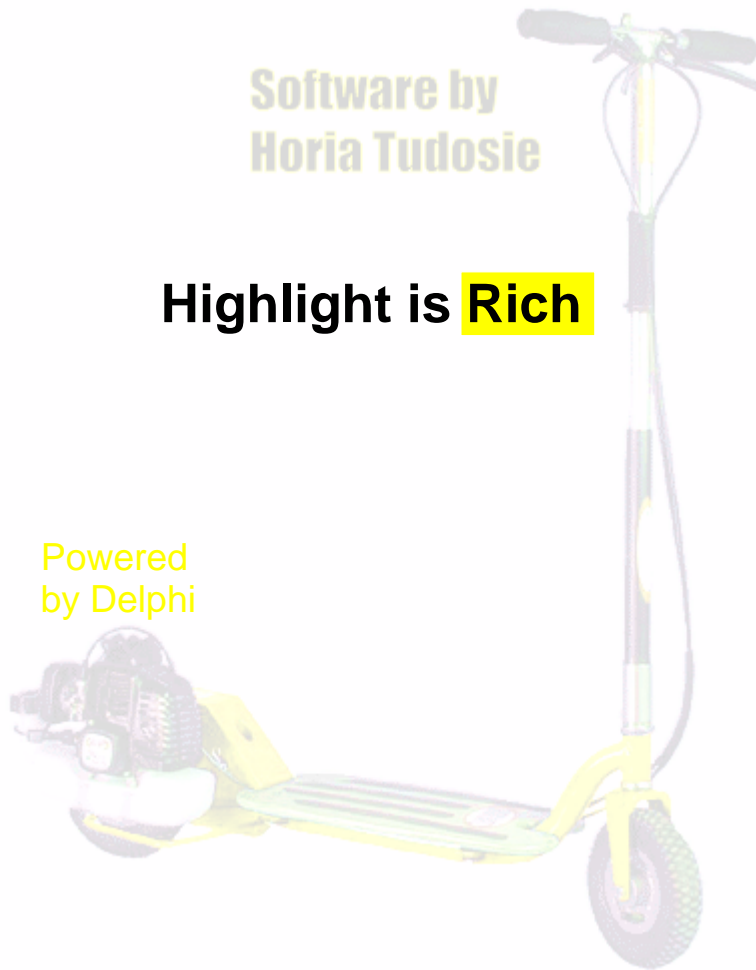


Software by  
Horia Tudosie

Highlight is Rich

Powered  
by Delphi



© 2005 - Horia Tudosie



---

# Table of Contents

<b>Highlight is Rich</b>	<b>2</b>
<b>Highlight Button</b> .....	<b>2</b>
<b>Add a menu</b> .....	<b>4</b>
<b>Make the menu works</b> .....	<b>8</b>
<b>Now at work</b> .....	<b>9</b>
The easy way: .....	9
The best way: .....	9
<b>Give it a Hint</b> .....	<b>10</b>
<b>Conclusion</b> .....	<b>10</b>

## Highlight is Rich

Did you ever notice that the `TRichEdit` component can change the color of the font for any series of characters but cannot change the color of the background of a portion of text? If you have a closer look to its properties you will find both `SelAttributes` and `DefAttributes` to be of type `TTextAttributes` having only a `Color` sub-property. Figure 1 lists the help for this property:

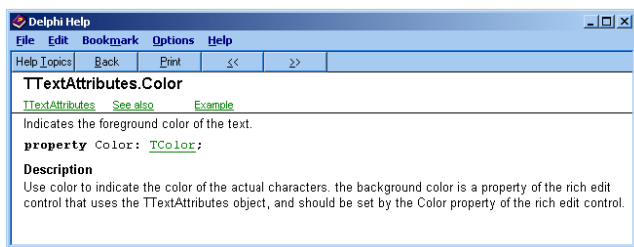


Figure 1

Looks like Delphi does not know that selected characters may have other color than the `Color` property of the rich edit control.

However, create a small document in Word, highlight some words with different colors, then cut and paste in a Delphi program containing a rich edit control – surprise: it works! `TRichEdit` control comes in Delphi from the `Win32` palette. This means that it is just a wrapper of a Windows control. If Office can highlight, Delphi also can by using a message:

```

1 uses RichEdit;
2
3 procedure SetSelBackgroundColor(
4   var RichEdit:TRichEdit; BackColor:TColor);
5   var Format: TCharFormat2;
6   begin
7     FillChar(Format, SizeOf(TCharFormat2), 0);
8     Format.cbSize := SizeOf(TCharFormat2);
9     with Format do begin
10      dwMask:= CFM_BACKCOLOR;
11      crBackColor:= BackColor;
12    end;
13    SendMessage(RichEdit.Handle, EM_SETCHARFORMAT,
14      SCF_SELECTION, LPARAM(@Format));
15 end;

```

Figure 2

`TCharFormat2` and the other constants are defined in the `RichEdit` unit, so include this unit in the `Uses` statement. Is not important for the

declaration part of the module, so add it in the implementation part of the code. The procedure may be added as a standalone procedure in an unit that serve any rich edit in the program, or may be bound to the `TForm` object witch contains the `RichEdit` to be served, or – alternatively – it may inherit from `TRichEdit` and build a new one with an added public method.

## Highlight Button

As you see, highlighting is no difficult – however this facility will enrich all your programs that let users see and use information in rich edit boxes.

So let's build a component – a button that is easy to use and deploy in any further programs. The button should have a glyph showing the selected highlight color and a way to change the selected color – a pop up menu. We should also be able to use it comfortably either by firstly selecting the text to highlight and then pressing the button, or by pressing the button and brushing the text.

So: go in the main menu and create a new component:

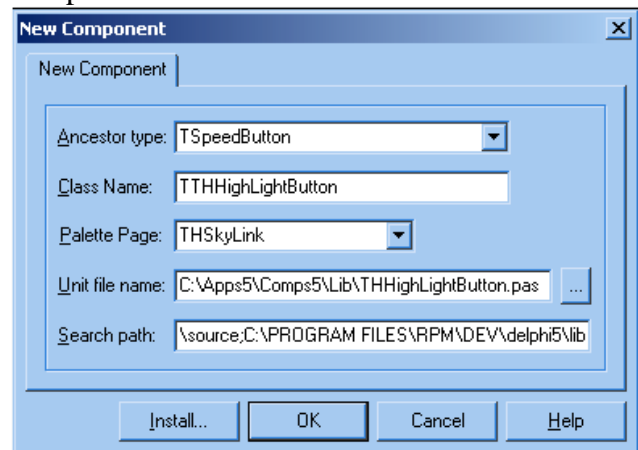


Figure 3

Add the `ComCtrls` unit in the `Uses` statement in the interface section and the `RichEdit` unit in the implementation section. Declare the `SetSelBackgroundColor` procedure and code it, so this unit will come in your program with the embedded procedure facilitating the desired

function for any RichEdit in the same form.

```

1  unit TTHighLightButton;
2
3  interface
4
5  uses
6    Windows, Messages, SysUtils,
7    Classes, Graphics, Controls,
8    Forms, Dialogs, Buttons, ComCtrls;
9  type
10   TTHighLightButton = class(TSpeedButton)
11   private
12     { Private declarations }
13   protected
14     { Protected declarations }
15   public
16     { Public declarations }
17   published
18     { Published declarations }
19   end;
20
21  procedure SetSelBackgroundColor(
22     var RichEdit:TRichEdit;BackColor: TColor);
23  procedure Register;
24
25  implementation
26
27  uses RichEdit;
28
29  procedure SetSelBackgroundColor(
30     var RichEdit:TRichEdit;BackColor: TColor);
31  var Format: TCharFormat2;
32  begin
33    FillChar(Format, SizeOf(TCharFormat2), 0);
34    Format.cbSize := SizeOf(TCharFormat2);
35    with Format do begin
36      dwMask:= CFM_BACKCOLOR;
37      crBackColor:= BackColor;
38    end;
39    SendMessage(RichEdit.Handle,
40      EM_SETCHARFORMAT,
41      SCF_SELECTION,LPARAM(@Format));
42  end;
43  procedure Register;
44  begin
45    RegisterComponents('THSkyLink',
46      [TTHighLightButton]);
47  end;
48  end.

```

Figure 4

The button will have an embedded glyph that will be visible both on the palette and on the button itself. Moreover, the button will provide a shape that will change the color accordingly with the selected color.

Open the Borland Image Editor, create a resource file and add a 16x16 bitmap:

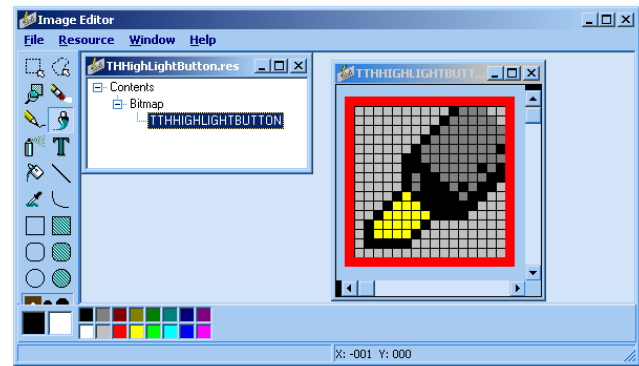


Figure 5

Name it **TTHIGHLIGHTBUTTON** and save the file as **TTHighLightButton.res** near the source code. Try to reproduce the above image and save it again. Add the `{ $R *.res }` line after the implementation statement and build the package again. This time, the component has the desired image on the palette, but still comes with no glyph on the form.

To have the glyph embedded in the component, override the **Loaded** method of the component:

```

1  procedure TTHighLightButton.Loaded;
2  var fGlyph:TBitmap;
3  begin
4    inherited;
5    fGlyph:=TBitmap.Create;
6    try
7      with TImageList.Create(nil) do
8        try
9          if ResourceLoad(rtBitmap,
10             'TTHIGHLIGHTBUTTON',
11             clSilver) then begin
12             GetBitmap(0,fGlyph);
13             Glyph.Assign(fGlyph);
14           end;
15         finally
16           free;
17         end;
18       end;
19     end;

```

Figure 6

The code in Figure 6 creates a Bitmap, an Image list, and uses the ResourceLoad method of the image list to retrieve the glyph from the palette. 'rtBitmap' is defined in the **ImgList** unit, so should be added it in the **uses** statement in the **implementation** section. Recompile the package and see the form: this time, the button

shows the embedded glyph.

Note: being defined in the `Loaded` method of the component, this code runs after the component has initialized the Glyph selected at design time. The embedded one will override it. It is possible to hide this property, but on a second thought: leave it there – somebody may want to read it later in the program, for example to assign it on another menu item in the main menu. Sometimes, when dropping the component from the palette, the glyph may not show. This is because the `Loaded` procedure does not fire when the component is brand-new on the form. To make the glyph visible, cut and paste back the component.

Now let's do something about the selected color. Let's define a new property `Color` and an event `OnColorChanged` that will notify the program when the color changes:

```

1 type
2   TTHighLightButton = class(TSpeedButton)
3   private
4     fColor:TColor;
5     fOnColorChanged:TNotifyEvent;
6   protected
7     procedure Loaded; override;
8     procedure SetColor(cl:TColor);
9   public
10  { Public declarations }
11  published
12    property Color:TColor
13      read fColor
14      write SetColor;
15    property OnColorChanged:TNotifyEvent
16      read fOnColorChanged
17      write fOnColorChanged;
18 end;
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

27 end;  
Figure 7

The `SetColor` procedure uses the passed color to paint directly on the glyph. To make sure the glyph shows the right color, this procedure is called in the end of the `'Loaded'` method. For this reason the procedure paints on the canvas of the glyph before testing that was a new color, but calls the notify event only then.

## Add a menu

Up to now, the component remains just a nice exercise. To make it useful, we should add an easy way to select a color. The most convenient way is to pop up a menu to select from. Pop up menus do not come easily in VCL components, but once embedded in the functionality of the components may encapsulate lots of features that no programmer will care to code directly on the glue logic of the form. For example, add an image showing the color of each item.

One problem with the popup menu of a component is that it cannot popup at design time. Another is that the programmer may want to add other items to the embedded menu. Also the menu should show witch color was selected before popping-up.

To solve all these, test the `ComponentState` and work on the menu only if the component is not in Design state. Do all processing in the end of the overridden `'Loaded'` method, so all other properties would be already initialized when starting building the menu.

Add a popup menu variable, some menu items and related methods in the protected section of the component (Figure 8.) Eventually other inherited components will use them.

```

1 protected
2   mclBlack :TMenuItem; mclMaroon :TMenuItem;
3   mclGreen :TMenuItem; mclOlive :TMenuItem;
4   mclNavy :TMenuItem; mclPurple :TMenuItem;
5   mclTeal :TMenuItem; mclGray :TMenuItem;
6   mclCustom1:TMenuItem; mclRed :TMenuItem;
7   mclLime :TMenuItem; mclYellow :TMenuItem;

```

```

8  mclBlue   :TMenuItem; mclFuchsia:TMenuItem;
9  mclAqua   :TMenuItem; mclSilver :TMenuItem;
10 mclWhite  :TMenuItem; mclCustom2:TMenuItem;
11
12 fPopupMenu :TPopupMenu;
13 fOldOnPopup :TNotifyEvent;
14 fImageList :TImageList;
15 fBitmap    :TBitmap;

```

Figure 8

`fPopupMenu` is a variable holding either a popup menu linked on the component at design time, either a new instance created inside the component. `TPopupMenu` and `TMenuItem` are defined in the `Menus` unit, so add it in the `uses` statement of the `interface` section. Each menu item will have nearby a small square showing the corresponding color. To achieve that, an image list should be associated with the popup menu component, 18 bitmaps should populate the list and each menu item should hold the index of the corresponding image in the list.

These menu items have to act as an array of radio buttons: click one of them, and the previously selected should automatically reset. For this, the menu item has to set the `RadioButton` property, and eventually all menu items should have the same `GroupIndex`. This is not necessary, but if the programmer wants to add other items to the built-in menu, then their `GroupIndex` should be not null. When `GroupIndex` is not null, it should be different than any other group of buttons or menu items in the form. Designing the component is not possible to know what other group indexes will be used in any arbitrary form. If the component won't expose this internal group index, no other programmer will be able to adapt or change it.

So, let's put all that in the constructor of the component (see Figure 9.)

```

1  constructor THighLightButton.Create(
      AOwner: TComponent);
2  var i:integer;
3  procedure NewMenuItem(
      var aMenuItem:TMenuItem;
      aColor:TColor;aCaption:string);
4  begin
5  fBitmap.Canvas.Brush.Color:=aColor;
6  fBitmap.Canvas.FillRect(Rect(0,0,16,16));
7  fImageList.Add(fBitmap,Nil);
8
9  aMenuItem:=TMenuItem.Create(Self);

```

```

10  aMenuItem.Caption:=aCaption;
11  aMenuItem.RadioItem:=true;
12  aMenuItem.GroupIndex:=fInternalGroupIndex;
13  aMenuItem.OnClick:=ColorClick;
14  aMenuItem.ImageIndex:=i;
15  i:=i+1;
16  end;
17 begin
18  inherited;
19
20  fInternalGroupIndex:=116;
21  fCustomColor1:=$00F0D7FF;
22  fCustomColor2:=$00D7FFF3;
23
24  fImageList :=TImageList.Create(nil);
25
26  fBitmap:=TBitmap.Create;
27  fBitmap.Height:=16;
28  fBitmap.Width :=16;
29
30  Color:=clYellow;
31
32  if (csDesigning in Owner.ComponentState) then
      exit;
33
34  i:=0;
35
36  NewMenuItem(mclRed, clRed, 'Red');
37  NewMenuItem(mclLime, clLime, 'Lime');
38  NewMenuItem(mclYellow, clYellow, 'Yellow');
39  NewMenuItem(mclBlue, clBlue, 'Blue');
40  NewMenuItem(mclFuchsia, clFuchsia, 'Fuchsia');
41  NewMenuItem(mclAqua, clAqua, 'Aqua');
42  NewMenuItem(mclSilver, clSilver, 'Silver');
43  NewMenuItem(mclWhite, clWhite, 'White');
44  NewMenuItem(mclCustom1,
      fCustomColor1,'Custom 1');
45  NewMenuItem(mclMaroon, clMaroon, 'Maroon');
46  NewMenuItem(mclGreen, clGreen, 'Green');
47  NewMenuItem(mclOlive, clOlive, 'Olive');
48  NewMenuItem(mclNavy, clNavy, 'Navy');
49  NewMenuItem(mclPurple, clPurple, 'Purple');
50  NewMenuItem(mclTeal, clTeal, 'Teal');
51  NewMenuItem(mclGray, clGray, 'Gray');
52  NewMenuItem(mclBlack, clBlack, 'Black');
53  NewMenuItem(mclCustom2,
      fCustomColor2,'Custom 2');
54 end;

```

Figure 9

The `NewMenuItem` procedure looks a little dirty. We could use lots of other ways to initialize the menu items, from direct programming 18 times the code in the procedure body, to inheriting from `TMenuItem` and redefining the constructor. However passing the `aMenuItem` variable as a `Var` parameter and using the global variable `i`, keeps the code balanced between elegance and efficiency.

`FBitmap` has to be created here and since it cannot be added in somebody's component list, we have to remember to free it in the destroyer. By contrast, `fImageList` may have an owner, but

creating it with no owner will make the code work faster and use less resources. Of course, we won't forget either to destroy it as we do with `fBitmap`.

```

1 procedure TTHHighLightButton.ColorClick(
  Sender: TObject);
2 var cl:integer;
3     s:string;
4 begin
5   with (Sender as TMenuItem) do begin
6     Checked:=true;
7     s:=(Sender as TMenuItem).Caption;
8     if pos('ustom',s)=0 then begin
9       while pos('&',s)>0 do
10        System.Delete(s,pos('&',s),1);
11      if IdentToColor('cl'+s,cl) then
12        Color:=cl;
13    else begin
14      if pos('1',s)>0 then
15        Color:=SelectColor(1)
16      else if pos('2',s)>0 then
17        Color:=SelectColor(2);
18    end;
19 end;
20 end;

```

Figure 11

There is no need to check the type of the `Sender` since this handler was not assigned to other type of components. However, typecast it as a menu-item type for accessing its menu-item properties. Setting the checked property automatically resets the previous selected menu item in the radio group. Setting the color does the rest in the `SetColor` method (Figure 7.) The only problem left is to find what color to assign on this property. For regular colors, just prefix the Caption with `'cl'` and pass the string to the `IdentToColor` Delphi function, but pay attention: Windows may insert the `'&'` character in menu captions, so line 9 in Figure 11 cares to remove any `'&'` character.

To separate custom colors from regular ones, search for the `'ustom'` string in the caption of the sender. The spelling `'ustom'` is not an error or an omission: in this situation, the system returns `'&Custom 1'` and `'&Custom 2'` as captions. Then `'1'` or `'2'` in the caption string may indicate the usage of `fCustomColor1` or `fCustomColor2` variables, but we want to do more: clicking on these two items will give the

choice of changing the custom color.

```

1 procedure SetCustomColor1(cl:TColor);
2 procedure SetCustomColor2(cl:TColor);
3 function SelectColor(i:integer):TColor;
4 published
5   property Color:TColor
6     read fColor write SetColor;
7   property OnColorChanged:TNotifyEvent
8     read fOnColorChanged write fOnColorChanged;
9   property InternalGroupIndex:integer
10    read fInternalGroupIndex
11    write SetInternalGroupIndex;
12   property CustomColor1:TColor
13     read fCustomColor1 write SetCustomColor1;
14   property CustomColor2:TColor
15     read fCustomColor2 write SetCustomColor2;
16 end;
17
18 function TTHHighLightButton.SelectColor(
19   i:integer):TColor;
20 var cl:TColor;
21     cd:TColorDialog;
22 begin
23   cl:=clNone;
24   case i of
25     1: cl:=CustomColor1;
26     2: cl:=CustomColor2;
27   end;
28   cd:=TColorDialog.Create(nil);
29   try
30     cd.Color:=cl;
31     cd.Options:=cd.Options+[cdFullOpen];
32     if cd.Execute then begin
33       case i of
34         1: CustomColor1:=cd.Color;
35         2: CustomColor2:=cd.Color;
36       end;
37     result:=cd.Color;
38   else result:=cl;
39   finally
40     cd.free;
41   end;
42 end;
43
44 procedure TTHHighLightButton.SetCustomColor1(
45   cl:TColor);
46 begin
47   fCustomColor1:=cl;
48   fBitmap.Canvas.Brush.Color:=cl;
49   fBitmap.Canvas.FillRect(Rect(0,0,16,16));
50   fImageList.Replace(8,fBitmap,Nil);
51 end;
52
53 procedure TTHHighLightButton.SetCustomColor2(
54   cl:TColor);
55 begin
56   fCustomColor2:=cl;
57   fBitmap.Canvas.Brush.Color:=cl;
58   fBitmap.Canvas.FillRect(Rect(0,0,16,16));
59   fImageList.Replace(17,fBitmap,Nil);
60 end;

```

Figure 12

To select a new color, create a color dialog (Figure 12) initialize it with the old color and fully open it – we do not expect the user to select

a predefined regular color! If it **execute** (the user has clicked OK - Figure 13) put the color back on the corresponding Custom Color property.

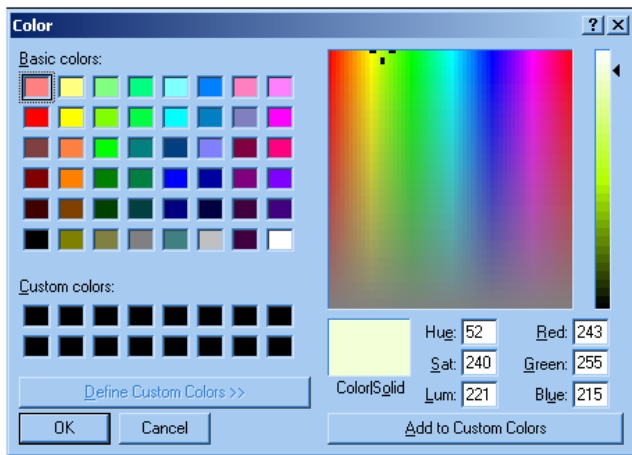


Figure 13 – Open a Color Dialog fully open.

It is not a good idea to put the new selected color directly on the corresponding `fCustomColorX` private variable. Instead let the `SetCustomColorX` methods serving the properties to do a little more: update the color in the menu by refilling the bitmap with the new color and replacing the corresponding bitmap in the image list.

To complete the code do the same with `SetInternalGroupIndex` – when the group index for the color menu-items changes, all menu items should be updated. Note that at design time this cannot happen, since the component does not create menu items when the program does not run. Also, the constructor already assigned the default internal group index. However, when the component loads, the value of the internal group index stored in the form may be different than the default one. Then, all menu items should be updated again.

```

1 procedure
TTHHighLightButton.SetInternalGroupIndex(
  i:integer);
2 begin
3   fInternalGroupIndex:=i;
4   if Assigned(mclCustom2) then begin
5     mclRed   .GroupIndex:=i;
     mclLime  .GroupIndex:=i;

```

```

6     mclYellow .GroupIndex:=i;
     mclBlue   .GroupIndex:=i;
7     mclFuchsia.GroupIndex:=i;
     mclAqua   .GroupIndex:=i;
8     mclSilver .GroupIndex:=i;
     mclWhite  .GroupIndex:=i;
9     mclMaroon .GroupIndex:=i;
     mclGreen  .GroupIndex:=i;
10    mclOlive  .GroupIndex:=i;
     mclNavy   .GroupIndex:=i;
11    mclPurple .GroupIndex:=i;
     mclTeal   .GroupIndex:=i;
12    mclGray   .GroupIndex:=i;
     mclBlack  .GroupIndex:=i;
13    mclCustom1.GroupIndex:=i;
     mclCustom2.GroupIndex:=i;
14  end;
15 end;

```

Figure 14

To find out if the menu items exist when changing the `InternalGroupIndex`, test if the last one – `mclCustom2` – was assigned, and only then changes their `GroupIndex` property. In this moment the code compiles, but has no effect! This is because the menu items are not used yet.

```

1 procedure TTHHighLightButton.Loaded;
. . .
2 if (csDesigning in Owner.ComponentState) then
3   exit;
4 if Assigned(PopupMenu) then begin
5   fPopupMenu:=PopupMenu;
6   fOldOnPopup:=fPopupMenu.OnPopup;
7   mclRed.Break:=mbBarBreak;
8 end
9 else begin
10  fPopupMenu:=TPopupMenu.Create(self);
11  PopupMenu:=fPopupMenu;
12 end;
13 mclMaroon.Break:=mbBarBreak;
14
15 fPopupMenu.Images:=fImageList;
16
17 fPopupMenu.Items.Add(mclRed);
18 fPopupMenu.Items.Add(mclLime);
19 fPopupMenu.Items.Add(mclYellow);
20 fPopupMenu.Items.Add(mclBlue);
21 fPopupMenu.Items.Add(mclFuchsia);
22 fPopupMenu.Items.Add(mclAqua);
23 fPopupMenu.Items.Add(mclSilver);
24 fPopupMenu.Items.Add(mclWhite);
25 fPopupMenu.Items.NewBottomLine;
26 fPopupMenu.Items.Add(mclCustom1);
27
28 fPopupMenu.Items.Add(mclMaroon);
29 fPopupMenu.Items.Add(mclGreen);
30 fPopupMenu.Items.Add(mclOlive);
31 fPopupMenu.Items.Add(mclNavy);
32 fPopupMenu.Items.Add(mclPurple);
33 fPopupMenu.Items.Add(mclTeal);
34 fPopupMenu.Items.Add(mclGray);
35 fPopupMenu.Items.Add(mclBlack);
36 fPopupMenu.Items.NewBottomLine;
37 fPopupMenu.Items.Add(mclCustom2);
38
39 fPopupMenu.OnPopup:=SpeedOnPopup;
40 end;

```

Figure 15

## Make the menu works

The best place to start using these menu items is in the `Loaded` method after is determined that the component is not in Design state. `Loaded` is fired after all initializations from to the form design ended, so the component knows exactly if it has or not a popup menu attached. Here the component may decide to add its menu items to a preexisting menu, or to create a new popup menu if the designer provided none.

These menu items being already constructed, a bar break can be added on the first menu item when a previous popup menu exists, and another bar break can be added anyway in the middle of the menu on the `Maroon` color menu item. Add a `NewBottomLine` before each `Custom X` color menu item.

Disregard/comment the line 39 in Figure 15, compile the package, put a `TTHighLightButton` on a form, run and right-click. The menu in Figure 16 pops up. Add a popup menu component on the form, add some items and refer it in the `PopupMenu` property of the button. Run and see the menu in Figure 17.



Figure 16

What's missing here (Figure 16) is the Yellow color menu-item being selected. Uncomment back the line 39 in Figure 15 and add the `SpeedOnPopup` method. Define it as `dynamic` in

the `protected` section of the component so an inherited component from this button could easily modify/replace it.



Figure 17 – after defining `SpeedOnPopup`.

```

1 procedure TTHighLightButton.SpeedOnPopup(
  Sender: TObject);
2 begin
3   if Color=clRed then
4     mclRed.Checked:=true
5   else if Color=clLime then
6     mclLime.Checked:=true
7   else if Color=clYellow then
8     mclYellow.Checked:=true
9   else if Color=clBlue then
10    mclBlue.Checked:=true
11  else if Color=clFuchsia then
12    mclFuchsia.Checked:=true
13  else if Color=clAqua then
14    mclAqua.Checked:=true
15  else if Color=clSilver then
16    mclSilver.Checked:=true
17  else if Color=clWhite then
18    mclWhite.Checked:=true
19  else if Color=clMaroon then
20    mclMaroon.Checked:=true
21  else if Color=clGreen then
22    mclGreen.Checked:=true
23  else if Color=clOlive then
24    mclOlive.Checked:=true
25  else if Color=clNavy then
26    mclNavy.Checked:=true
27  else if Color=clPurple then
28    mclPurple.Checked:=true
29  else if Color=clTeal then
30    mclTeal.Checked:=true
31  else if Color=clGray then
32    mclGray.Checked:=true
33  else if Color=clBlack then
34    mclBlack.Checked:=true
35  else if Color=CustomColor1 then
36    mclCustom1.Checked:=true
37  else begin
38    CustomColor2:=Color;
39    mclCustom2.Checked:=true;
40  end;
41 end;
42 if Assigned(fOldOnPopup) then
43   fOldOnPopup(sender);
44 end;

```

Figure 18

The procedure compares the color against all menu colors and selects the corresponding menu item. If the color does not match a menu-item color, then it changes the `CustomColor2` and selects its menu item.

In the end the procedure calls the `OnPopup` handler of the previous menu if it was one. Figure 17 shows how the Yellow menu item appears as selected.

This concludes our new component – `TTHighLightButton`. See the final code in Figure 22.

## Now at work

There is an easy way and a hard way:

### The easy way:

If you do not have yet a rich edit component on the test form, put one now. Then on the `OnClick` event of the `TTHighLightButton1` add the code from Figure 19.

```
1 procedure TForm1.TTHighLightButton1Click(
  Sender: TObject);
2 begin
3   if RichEdit1.SelLength=0 then exit;
4   SetSelBackgroundColor(RichEdit1,
  TTHighLightButton1.Color);
5   RichEdit1.SelStart:=
  RichEdit1.SelStart+RichEdit1.SelLength;
6 end;
```

Figure 19

If nothing is selected in the rich edit text, the procedure just exits. Else, it calls the `SetSelBackgroundColor` standalone procedure from the `TTHighLightButton` unit.

Using this method implies that the user has first selected the desired portion of text, and then clicks the button. When there are lots of items to highlight, that is not very handily.

### The best way:

Put a non-null value on the `GroupIndex` property of the button and set its `AllowAllUp` property.

This will make the button having two states: pressed and depressed. When the button is pressed, whatever we highlight automatically changes the color with the selected color of the button. To do that use the `OnMouseUp` event of the rich edit component.

```
1 procedure TForm1.TTHighLightButton1Click(
  Sender: TObject);
2 begin
3   if (RichEdit1.SelLength<>0)
4   and TTHighLightButton1.Down then begin
5     SetSelBackgroundColor(
  RichEdit1,TTHighLightButton1.Color);
6     RichEdit1.SelStart:=
  RichEdit1.SelStart+RichEdit1.SelLength;
7     RichEdit1.SelLength:=0;
8     TTHighLightButton1.Down:=false;
9   end;
10 end;
11
12 procedure TForm1.RichEdit1MouseUp(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
13 begin
14   if TTHighLightButton1.Down then begin
15     TTHighLightButton1Click(Sender);
16     TTHighLightButton1.Down:=(ssAlt in Shift);
17   end;
18 end;
19 end;
```

Figure 20

There are three ways to exploit this code:

1. Highlight some text in the rich edit.

`RichEdit1MouseUp` does not fire inside because `TTHighLightButton1.Down` is `false`. Now click the button.

`TTHighLightButton1Click` fires, because there is some text highlighted, and the button went down. After the standalone procedure `SetSelBackgroundColor` finished its job, it removes the highlighting of the text leaving the cursor in the end of the highlighted zone, and releases the button. This acts like the previous example.

2. Press the button. Because nothing is selected, the button stay pressed. Now highlight some text. When releasing the mouse, `RichEdit1MouseUp` fires calling `OnClick` event of the button. The `OnClick` handler releases the button, but `RichEdit1MouseUp` corrects this using the A key state. Because you didn't press this key, the button releases itself.
3. Do the same as previous exercise, but press the

A key, and keep it down. The line 17 in Figure 20 puts the button down allowing continuing highlighting other selections, until the A key will be released. Exit the highlight mode either by releasing the A key before finishing the last selection, either by clicking again the button.

What else would somebody expect more? – oh, yes! – A hint!

## Give it a Hint

```

1 procedure TForm1.
  THHighLightButton1ColorChanged(
    Sender: TObject);
2 var Ident:string;
3 begin
4   if ColorToIdent(
    THHighLightButton1.Color,Ident) then
5     Ident:=copy(Ident,3,Length(Ident))
6   else
7     Ident:='$'+
      IntToHex(THHighLightButton1.Color,6);
8   THHighLightButton1.Hint:=
    'Highlight selections in the text with a '+
    Ident+' color.'^M^J+
    'Hold the Alt Key for multiple selections.';
9   THHighLightButton1.ShowHint:=true;
10  end;
11
12
13
14 procedure TForm1.FormCreate(Sender: TObject);
15 begin
16   THHighLightButton1ColorChanged(Sender);
17 end;

```

Figure 21

It is possible to code the hint inside the component. However that would have restricted the user to use its own wording or style.

The `THHighLightButton1ColorChanged` procedure uses the `ColorToIdent` Delphi function to find out if the color is a basic one and to decode its name. For `Custom 1` and `2` colors it just decode them in hex base.

`THHighLightButton1ColorChanged` does not automatically fire when the program starts, so call it when the form is created .

## Conclusion

The current tutorial shows how to build a new button that initialize its glyph and change it

accordingly with its status. It also presents a technique to add an embedded popup menu in a component; the technique allows the user to add other items in the embedded popup. The component code manipulates bitmaps and associates them with the menu items in the popup menu. Finally, the component commands a color-highlight function that extends the functionality of a rich edit component.

# Index



AllowAllUp 9



Bitmap 2, 4  
Borland Image Editor 2  
Button 2



Canvas 2  
| Brush 2  
| Pen 2  
Checked 8  
Color 2  
Color Dialog 4  
ColorClick 4  
ColorToIdent 10  
ComCtrls 2  
Component 2  
ComponentState 4



DefAttributes 2  
Down 9  
Dynamic 8



FormCreate 10



Glyph 2  
GroupIndex 4  
GroupIndex 9



Hint 10



IdentToColor 4  
ImageList 4  
ImgList 2  
InternalGroupIndex 4



Loaded 2, 4, 8



MenuItem 4



NewBottomLine 8  
NewMenuItem 4  
NotifyEvent 4



Office 2  
OnClick 9  
OnMouseUp 9  
OnPopup 8



Palette 2  
PopupMenu 4, 8  
Protected 8



RadioButton 4  
| GroupIndex 4  
Register 2  
ResourceLoad 2

RichEdit1MouseUp 9



SelAttributes 2  
SelLength 9  
SelStart 9  
SetColor 2  
SetSelBackgroundColor 2, 9  
SpeedOnPopup 8



TBitmap 4  
TColorDialog 4  
| Create 4  
| Execute 4  
TForm 2  
THighLightButton1Click 9  
THighLightButton1ColorChanged 10  
TImageList 4  
TNotifyEvent 4  
TPopupMenu 4  
TRichEdit 2  
TTextAttributes 2  
| DefAttributes 2  
TTextAttributes.SelAttributes 2  
TTHighlightButton 9  
| .res 2



Uses 2  
| ComCtrls 2  
| ImgList 2  
| Menus 4



Win32 2