Document :: Dialog Boxes2, Creating Your Own
Author :: Jon Jenkinson, jon.jenkinson@mcmail.com

From "The Bits" Website
http://www.cbuilder.dthomas.co.uk
email : forgot@mcmail.com

Legal Stuff

Dialog Boxes

This tutorial takes over from the first in the series, “Dialog Boxes Part One”, which built a basic text editor to show the built in features of standard dialogs. If you haven’t read Part One, do so as I assume you will already have a part completed program at this point.

Colour Coding:

```
Information displayed in this colour is information provided by my copy of C++ Builder.
Information displayed in this colour is information you need to type.
Information displayed in this colour is information which is of general interest.
```

****

Pre-Requisites

That you have completed part one:)

How to do Create Your Own Dialogs

Step One : Understanding Sets.

As I mentioned at the beginning of part one, all windows can be counted as a form of dialog, and as such, it should come as no surprise to find that we will use normal forms to create our own dialogs. Our first dialog will be a simple one. It will extend our little editor to allow the user to enter specific combinations of text style to a selection, and we will fire it from a keypress to make it a quick and easy interface.

Before actually designing any dialog, it is important to decide what information you need to get from the user, and just importantly, what information you're required to offer the user to enable them to tell you what they want.

In this case, we need to know how the RTF component handles the style attributes we're interested in. We will implement four possible single *styles*. They are, **bold**, *Italic*, Underline and ~~StrikeOut~~. These values, as mentioned earlier, are held in a set, and as such, here is a quick description of sets.

What is a set. Simply put it is a container, which can hold values of the same data type. Whilst these data types can have different names, they usually evaluate to an int. (e.g. char, int, enum). A set works on the following principal,

(a) Declare the type of set and give it a name
(b) Create an *instance* of the set
(c) Add values to, or remove values from the set.

Okay, here is a set definition,

```
typedef Set <int, 10, 20> RangeSet;
```

What we've done here is create a set *Type*, called RangeSet. RangeSet is a *Set Type* which is capable of holding integers with a minimum value of 10, and a maximum value of 20.

Now we shall create two instances of our set, called range1 and range2,

```
RangeSet range1, range2;
```

This gives us two *empty* sets. An empty set is simply a set which doesn't have any values, and is important to us later when we are comparing sets. To describe our sets in mathematical terms, *(yes they were a mathematicians invention:)*, we would write the following,

```
range1 = {}
range2 = {}
```

Simple:) Now let's assign some values to our sets,

```
range1 << 12 << 12 << 13 << 18;
range2 << 17 << 18 << 19 << 20;
```

Our sets now contain the following,

```
range1 = {12, 13, 18}
range2 = {17, 18, 19, 20}
```

Pay attention to this point, as it is very important to the way we use sets in programming. A set can only contain at most, one occurrence of a given value. Thus, this simplifies our handling of sets. If our user wishes to assign a value to a set, we **do not** need to check if that value is already there, we just assign it again, if it's there already, it doesn't matter, and if it isn't, it is now:)

To remove a value from a set, we would do the following,
range1 >> 17 >> 12 >> 20;
range2 >> 17 >> 12 >> 20;

Now our sets contain the following values,
range1 = {13, 18}
range2 = {18, 19}

Notice two things here. When we remove 12 from our first set, it removes the occurrence of 12, re previous point. Secondly, it doesn't matter whether the set contains the value we wish to remove or not, if it's there, we remove it, if it isn't, it simply doesn't matter.

Next, we'll looking at assigning and removing values outside our range,
range1 << 3 << 4 << 5;
range2 >> 27 >> 50 >> 19;

The only thing in the above two statements which would have any effect is the removal of 19 from range2. As the other values are outside the defined min max values of the set, they are ignored.

A word of warning, the following throws a compiler warning, and it is unclear what will happen at runtime. Whilst possible to get around this statement using brackets, it is clearer and better practice to split it into two statements, one for the insertions, (<<), and one for the removals, (>>).
range2 << 24 >> 29 << 19 >> 18 << 17;
should become
range2 << 24 << 19 << 17;
range2 >> 29 >> 18;

Before we move on, I'll introduce the actual font->Style set we'll be dealing with, TFontStyles

TFontStyles is a typedef of the *TFontStyle* enum. This is declared in the C++ Builder headers as follows,

enum TFontStyle { fsBold, fsItalic, fsUnderline, fsStrikeOut };
typedef Set<TFontStyle, fsBold, fsStrikeOut> TFontStyles;

What this means is actually very simple. Firstly we enumerate the TFontStyle, giving the following values,
fsBold = 0
fsItalic = 1
fsUnderline = 2
fsStrikeOut = 3

We then declare a *set type* capable of holding types of TFontStyle, within the min max range of fsBold to fsStrikeOut, (0-3). Remember that our set can also hold the empty set value, and you get a possible combination of a permutation of five values. If you tried to handle that permutation with a switch or if tree, you'd have a lot of code to write. With sets, we can simply work with each possible *element*, and handle things an element at a time, without concern for the rest of the set contents.

The final thing to learn about sets is the *intersection*, *difference*, *union* and the set operators. We've met two of the operators already, << and >>, for adding and removing from sets. The other true operators are the equality and assign operators.

To perform an equality test and an assignment, the two sets are required to be of the same type. Therefore, when declaring sets for these tests, you must use the same type. The following line of code will create four sets of the type we are interested in.

TFontStyles MyFontStyle1, MyFontStyle2, MyEmptyFontSet, TheCurrentFontSet;

We now have four sets of the same type, so let's assign values to them,

MyFontStyle1 << fsBold << fsItalic;
MyFontStyle2 << fsUnderline << fsStrikeOut;
TheCurrentFontSet << fsBold << fsItalic << fsUnderline;

Our sets now contain the following values,

MyFontStyle1 = {fsBold, fsItalic}
MyFontStyle2 = {fsUnderline, fsStrikeOut}
TheCurrentFontSet = {fsBold, fsItalic, fsUnderline}
MyEmptyFontSet = { }

The equality test, being performed on sets of the same type can now be used.

if(MyFontStyle1 == TheCurrentFontSet) would return false.

You can assign a set to be a copy of a current set using the assignment operator, =. The following creates a new set, temp, an exact copy of the current state of MyFontStyle1,

TFontStyles temp = MyFontStyle1;

Then the following equality test would be true,

if(MyFontStyle1 == temp).

Not that we'll use it here, but the remaining three set qualities are as follows,

Intersection

The intersection is just that, the values that appear in both of two sets,
TFontStyles temp = MyFontStyle1 *TheCurrentFontSet;

would produce a set temp with the following elements,
temp = {fsBold, fsItalic},

that is the values which appear in both of our sets. Note the return of a set as the answer

Difference

Again, just that, the values that appear in one set, but not the other
TFontStyles temp = MyFontStyle1 - TheCurrentFontSet;

would produce a set temp with the following elements
temp = {fsUnderline}

Union

The union of two sets, is a set which contains the values which appear in either set,
TFontStyles temp = MyFontStyle1 + TheCurrentFontSet;

would produce a set temp with the following elements
temp = {fsBold, fsItalic, fsUnderline}.

Right, there is one function of sets which I haven't mentioned till now, and it is the most important to our use of sets in this instance. It is the Contains(x) function. What this does is tell us whether the set *contains* the value x or not. It doesn't cause an error if the value of x is outside the range of our set.

if(MyFontStyle1.Contains(fsBold)) would return true.

Step Two :Okay, lets make our dialog

Now we know what we're doing with the font->style set, let's build our dialog. Choose File|New Form, and you get Form2 and Unit2.cpp. Let's start to make our project more sensible, and save Unit2.cpp as OurDialog.cpp with File|Save As.

Now that's done let's look at the form itself. During design, the appearance of the form wont change, but we do need to make some changes if we're to make it into a dialog. Set the form's properties as follows,

BorderIcons
    All False
BorderStyle = bsDialog
Caption = "Style Selection"
FormStyle = fsStayOnTop#
Name = StyleChoice
Position = poScreenCenter
Visible = false

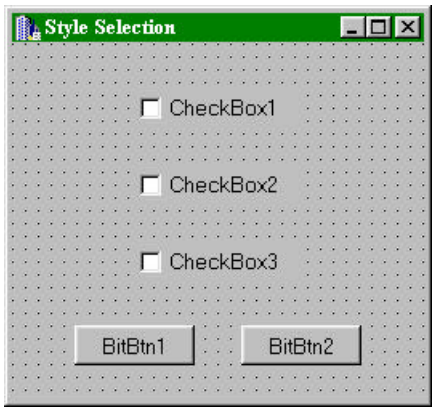
The rest of the options should be left as their default values, with the exception of WindowState, make sure it is set to wsNormal.

Now we actually have a dialog form, but it doesn't do anything. Whilst the actual design of the Dialog is personal, lay it out using the following, and you should end up with something akin to the picture below,
(Picture A).

Start by placing three CheckBox's on the form. Next place two BitBtn's on the form from the Additional tab. Select the three CheckBox's and align them using the align dialog, (right click align), to Horizontal = Center In Window, Vertical = Space Equally.

Select the two BitBtns and align them to Horizontal = Center In Window.

Your form should look something like this,



Picture A, Our Basic Dialog Form.

If it doesn't, don't worry, as long as you've got the components on the form and it looks reasonable. Note that I've resized the form by eye to an approximate size:)

Now select CheckBox1, and set its properties as follows,

Caption = &Bold
Name = BoldCheck

Select the second check box and set the same properties

Caption = &Italic
Name = ItalicCheck

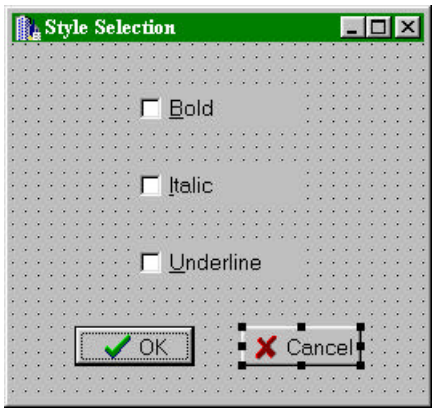
And the third,

Caption = &Underline
Name = UnderCheck

Before moving on, for those who hadn't noticed this feature, note how the use of the & creates a hotkey to the checkbox. This allows the user to select our checkbox by using the standard Windows method of Alt+underlined letter that they are used to:)

Now onto the buttons, BitBtn1 set the kind property to bkOK and BitBtn2 set the kind property to bkCancel.

Your form should now look something like the following,



Picture B, Our finished form.

So let's start to write the code we need to operate our dialog.

We could make life a lot easier here by adding an option to our menu, say Format, or a button bar with Ctrl+B for bold etc, but remember we're looking at dialogs. You'll have seen in other applications that you frequently have a menu as we have, but that you can fire events using shorthand key combinations, such as Ctrl+S for save, Ctrl+O for open etc.

We can do the same within C++ Builder, using a function built into Builder's forms. Bring up Form1 in the Object Inspector, and look for the property called "KeyPreview". It is currently set to false, which is no use to us, so let's set it to true.

The KeyPreview property basically says that the form will preview **ALL** keypresses before passing them to the ActiveControl. In our little example, we wish to be typing in our RTF and then hit, er..Ctrl+T to bring up our dialog. (*That's T for format if you were wondering:)*. Later we'll extend this to provide the standard methods, Ctrl+B for bold etc.

Okay, the event we need, is the form's OnKeyDown handler, and after you've added the code in red, it should look like this,

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
  TShiftState WeWantThis;

  WeWantThis << ssCtrl;

  if(Shift == WeWantThis)
  {//The Ctrl Key Has Been Pressed
    Form1->Caption = "Ctrl is down";
  }
  else
  {
    Form1->Caption = "Who Cares:)";
  }
}
```

Now this is a big leap in the imagination, so we'll take it one step at a time. Run the app, and press the control key. You'll notice that the caption of the form changes to tell you the control key is down. Type another key, and it changes to the second caption. Now type Ctrl+T together, and notice two things, as soon as you've pressed the control key the caption changes, and, more importantly, the T is intercepted, it doesn't appear on the form.

Okay, now you see how it works, aren't you glad I waffled on about sets so much earlier. The Shift state is passed as a set, allowing multiple key/mouse states to be checked for. The possible set members are,

ssShift: //The Shift key is held down.
ssAlt: //The Alt key is held down.
ssCtrl: //The Ctrl key is held down.
ssLeft: //The left mouse button is held down.
ssMiddle: //The middle mouse button is held down.
ssDouble: //Both the right and left mouse buttons are held down.

To capture this, we build a set of our own called WeWantThis, with the single member ssCtrl. The reason for doing it this way is that we are interested ONLY in the Ctrl key. If we were to try the set function, .Contains(ssCtrl), for instance, we would intercept the passed set which contains ssCtrl, whether or not the set contained other members. ***We Don't Want this.*** If the user presses another shift state key, we don't care, we only want it if it the passed set contains

ssCtrl exclusively. Test this out, press Shift+Ctrl together, and note that our set caption ignores it. Even if you press the Ctrl key first, as soon as you press the shift key, we ignore it:)

Right, if that's clear we'll move on, if you're unsure, just check that last bit again. The Key, how do we find out which key has been pressed in combination with our Ctrl Key. Change the handler to the following,

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
  TShiftState WeWantThis;

  WeWantThis << ssCtrl;

  if(Shift == WeWantThis)
  {//The Ctrl Key Has Been Pressed
    switch(Key)
  {
    case 'T':
    case 't':
      //we want this
      Form1->Caption = "Our key combo has been pressed";
      break;
    default:
      //ignore all
      Form1->Caption = "It isn't ours";
      break;
    }//end switch
  }
  else
  {
    Form1->Caption = "It's a normal key:)";
  }
}
```

Okay, again a little bit of playing here. Run the application and press the Ctrl key. The caption tells you it isn't ours. Press the T key, and we've caught it. Press anything else, and we treat it as a normal key, even if it's another state key, it's of no interest to us.

For your information, the non-alphanumeric keys have the following values if you wish to intercept them, note, unlike catching a normal alphanumeric, you don't include it in character quotes, that is VK_F1, not 'VK_F1'. Finally, note that some of the combinations, whilst trapped, continue with the default processing. e.g. pressing Ctrl+Tab will be caught, but will still put a tab in the rich edit. I have highlighted these keys in RED

(***The constants listed in this colour have not been tested, as I do not know which keys they are. They are provided for completeness only:)***

Symbolic constant name	Mouse or keyboard equivalent
VK_CANCEL	Control-break processing
VK_BACK	BACKSPACE key
VK_TAB	TAB key
VK_CLEAR	CLEAR key
VK_RETURN	ENTER key
VK_PAUSE	PAUSE key
VK_SPACE	SPACEBAR
VK_PRIOR	PAGE UP key
VK_NEXT	PAGE DOWN key
VK_END	END key
VK_HOME	HOME key
VK_LEFT	LEFT ARROW key
VK_UP	UP ARROW key
VK_RIGHT	RIGHT ARROW key
VK_DOWN	DOWN ARROW key
VK_SELECT	SELECT key
VK_EXECUTE	EXECUTE key
VK_INSERT	INS key

```
VK_DELETE                     DEL key
VK_HELP                       HELP key
VK_LWIN                       Left Windows key (Win95 Keyboard)
VK_RWIN                       Right Windows key (Win95 Keyboard)
VK_APPS                       Applications key (Win95 Keyboard)
VK_MULTIPLY                   Multiply key , (numeric keypad not Shift 8:)
VK_ADD                        Add key, (numeric keypad not Shift =:)
VK_SEPARATOR                  Separator key
VK_SUBTRACT                   Subtract key, (numeric keypad not _- key:)
VK_DECIMAL                    Decimal key, (numeric keypad not >. key:)
VK_DIVIDE                     Divide key, (numeric keypad not ?/ key:)
VK_F1   to VK_F24             F1 through to F24,  (whatever F24 is:)
VK_ATTN                       Attn key
VK_CRSEL                      CrSel key
VK_EXSEL                      ExSel key
VK_EREOF                      Erase EOF key
VK_PLAY                       Play key
VK_ZOOM                       Zoom key
VK_NONAME                     Reserved for future use.
VK_PA1                        PA1 key
VK_OEM_CLEAR                  Clear key

The following keys cannot be caught by this method as they do not produce a kepress
event:(
VK_ESCAPE                     Escape Key
VK_CAPITAL                    Caps Lock Key
VK_SNAPSHOT                   PRINT SCREEN key for Windows 3.0 and later
VK_NUMLOCK                    NUM LOCK key
VK_SCROLL                     SCROLL LOCK key
```

Okay, back to our example:)

We can now catch the Ctrl+T keypress, so let's clean up the handler and finally fire up our dialog box. Change the handler to the following,

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
  TShiftState WeWantThis;

  WeWantThis << ssCtrl;

  if(Shift == WeWantThis)
  {//The Ctrl Key Has Been Pressed
    switch(Key)
    {
      case 'T':
      case 't':               //we want this
        StyleChoice->ShowModal();
      break;
        default:              //ignore all
      break;
    }//end switch
  }
}
```

Include OurDialog.h at the top of Unit1.cpp. Nice and simple. Run the application, press Ctrl+T and it fires our dialog box. Note, within this function, we are not interested in the response from the dialog. That handling will be done within the dialog unit itself.

Before moving on, let's update the above handler to allow our user to access the normal keypresses for the dialogs our application already contains. Note how we simply call the menu onclick handler here, rather than rewriting the code and separate handlers.

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,TShiftState Shift)
{
  TShiftState WeWantThis;

  WeWantThis << ssCtrl;
```

```
  if(Shift == WeWantThis)
  {//The Ctrl Key Has Been Pressed
    switch(Key)
    {
      case 'T':
      case 't':               //we want this
        StyleChoice->ShowModal();
        break;
      case 'S':
      case 's':
        SaveClick(Sender);
        break;
      case 'P':
      case 'p':
        PrintClick(Sender);
        break;
      case 'O':
      case 'o':
        LoadClick(Sender);
        break;
      case 'F':
      case 'f':
        FindClick(Sender);
        break;
      default:         //ignore all
        break;
    }//end switch
  }
}
```

Finally for our first dialog, let's make it do something.

Include Unit1.h at the top of OurDialog.cpp, and then place the following code in StyleChoice's OnShow handler. (*Note: we place it in the OnShow so that it updates every time we fire the dialog:)*

```
void __fastcall TStyleChoice::FormShow(TObject *Sender)
{
  if(Form1->REdit->SelAttributes->Style.Contains(fsBold))
  {
    BoldCheck->State = cbChecked;
  }

  if(Form1->REdit->SelAttributes->Style.Contains(fsItalic))
  {
    ItalicCheck->State = cbChecked;
  }

  if(Form1->REdit->SelAttributes->Style.Contains(fsUnderline))
  {
    UnderCheck->State = cbChecked;
  }
}
```

And in BltBtn1's OnClick handler,

```
void __fastcall TStyleChoice::BitBtn1Click(TObject *Sender)
{
  switch(BoldCheck->State)
  {
    case cbUnchecked:
      Form1->REdit->SelAttributes->Style =
      Form1->REdit->SelAttributes->Style >> fsBold;
      break;
    case cbChecked:
      Form1->REdit->SelAttributes->Style =
      Form1->REdit->SelAttributes->Style << fsBold;
      break;
    default:   //Do Nothing
      break;
  }//end bold switch
```

```
  switch(ItalicCheck->State)
  {
    case cbUnchecked:
      Form1->REdit->SelAttributes->Style = Form1->REdit->SelAttributes->Style >> fsItalic;
      break;
    case cbChecked:
      Form1->REdit->SelAttributes->Style = Form1->REdit->SelAttributes->Style << fsItalic;
      break;
    default:   //Do Nothing
      break;
  }//end italic switch

  switch(UnderCheck->State)
  {
    case cbUnchecked:
      Form1->REdit->SelAttributes->Style = Form1->REdit->SelAttributes->Style >>
        fsUnderline;
      break;
    case cbChecked:
      Form1->REdit->SelAttributes->Style = Form1->REdit->SelAttributes->Style <<
        fsUnderline;
      break;
    default:   //Do Nothing
      break;
  }//end underline switch
}
```

Note that in the button's on click handler we have to take account of the fact that we need not only to add the style, but also remove it.

Run the application and test out your dialog. Set some styles, and then unset them. Set them via the font dialog we used earlier, and check that the currently applied styles appear when you launch your style dialog.

Step Three : "War & Peace?", You ain't seen nothing yet:)

If you thought we'd covered all about dialog boxes, you honestly haven't seen anything of their real power yet. Have a break, and then read on.

All applications have About Boxes, indeed a very basic one is built into C++ Builder. However, I prefer one which looks a little more professional, and as you've seen so far, a dialog is nothing more than a window, so lets make a very simple, but effective About Box for our application.

We need another form, so File|New Form, and save it as aboutbox.cpp using File|Save As.

Now with this dialog we're going to use some of those compenents which you wouldn't really associate with dialogs, Graphics Files, Tabbed Notebooks, and changing buttons.

Okay, set your the forms properties as follows, (this is the form probably called Form2 at present).

BorderIcons
        All false,
BorderStyle = bsDialog
Caption = AboutBox
FormStyle = fsStayOnTop
Position = poScreenCenter
Visible = false

Once you've done that, drop a standard panel on the form, and set it up as follows,

Align = alBottom
Alignment = taLeftJustify
BevelInner
BevelOuter both to bvLowered
Caption = “ My Clever About Box” //note the spaces, you can call it something different:)
Font->Color = clActiveCaption //personal, but I think it looks good:)
Name = NameBox

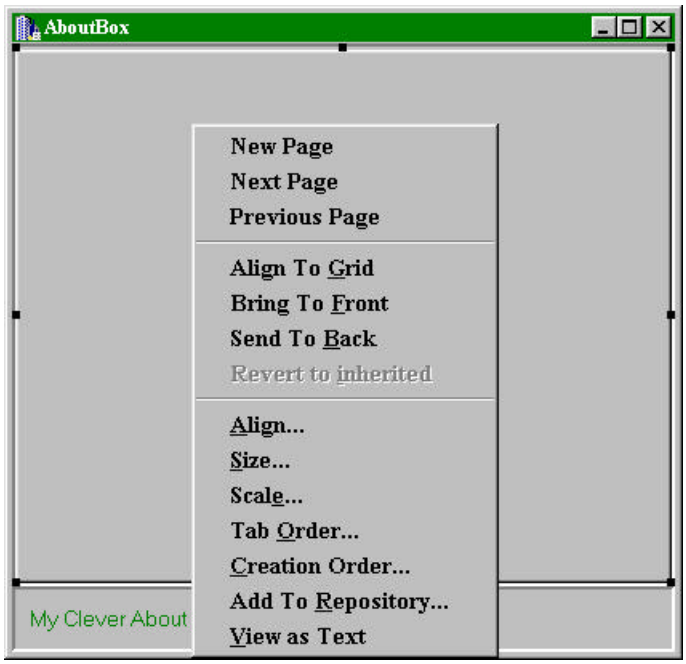
Drop another panel in the middle of the form, and set

Align = alClient
BevelInner and BevelOuter = bvLowered

Next drop a TPageControl on your form from the Win95 tab, and set its properties as follows

Align = alClient
Name = MasterPage

Now right click on the *MasterPage* on the form, and you get the following pop up menu,



Picture C, The Right Menu Specific To PageControls

Click on Add Page, then select the Page Control again and add another page, until you have six pages in all.

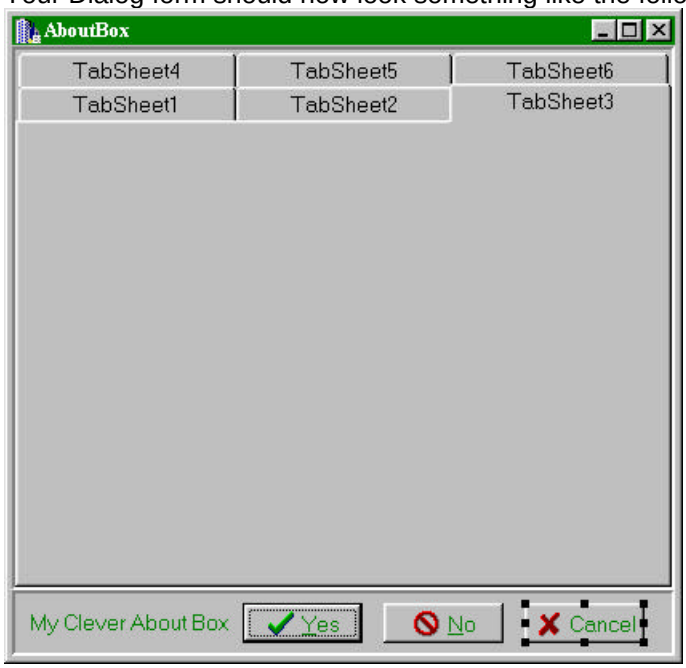
Notice that on doing that the PageControl adds two arrows at the top right. I prefer the *MultiLine view*, which is achieved by setting the PageControl’s MultiLine property to true, but it’s personal so I’ll leave it up to you.

Finally, add three BltBtns to the right of your caption in the bottom panel, and set them up as follows,

BltBtn1
Name = Butt1

Kind = bkYes
BltBtn2
Name = Butt2
Kind = bkNo
BltBtn3
Name = Butt2
Kind = bkCancel

Your Dialog form should now look something like the following,



Picture D, All six Tabs In Place

Okay, lets start by adding somethings to our dialog. Let's do the main thing, make it a true about box. Select TabSheet1 by clicking on the Tab at the top, and then in the middle of the dialog. Set the Caption of the Tab to "&About" and the Name to About. Place a TImage on the left hand side of the page, and a RichEdit on the right hand side, setting dimensions to suit your eye.

Change the font colour of the buttons, if you wish, I've left them green here, but I changed them to black later:)

Load a bitmap into the TImage, and set the Stretch property to true, and then set the RichEdit up as follows,

Alignment = taCenter
Color = clBtnFace
Enabled = false
ReadOnly = true
TabStop = false

and finally set the Lines up using the ellipse and editor. Your box should look like this,

"

About Our Little Program

Version 0.0a

A Simple Little Editor
Which Shows off Dialog Boxes

©The Bits They Forgot!
C++ Builder Pages"

Next, add a menu option called About, with the caption &About to our menu on form1, and then add the following handler to that menu's OnClick event,

```
void __fastcall TForm1::AboutBoxClick(TObject *Sender)
{
  Form2->MasterPage->ActivePage = Form2->About;
  Form2->Caption = "About our little program";
  Form2->Butt1->Visible = false;
  Form2->Butt2->Visible = false;
  Form2->Butt3->Kind = bkClose;
  Form2->ShowModal();
}
```

Now run the application, choose the about box, Alt+A, and you should get something which looks similar to the following,

Picture E, Our running About Box

The thing to notice in this little box is that the buttons can be changed, hidden etc, and that although you can put the cursor in the RichEdit, you can't change it. We've changed the caption of our form, and we could actually change anything, it isn't hard wired.

Step Four : Getting Really Clever:)

This leads to the final part of the tutorial about dialogs, *(aren't you glad:)*. You can do many of the easy things with the other tabs here, place an RichEdit on one and read your information in

from a file, even make one into a simple help system if you wish, but we're going to build a tab which reads some information from the system, and then allows the user to *Register* our little product, for now just by writing it out to a file, but we could fire an e-mail etc.

This shows a much underused principal when it comes to dialogs. As they are simply windows, you can treat them as a single resource. By this I mean that no dialog is fixed. If you have a dialog for input, say of a telephone number, you can *reuse* that dialog for inputting a name or address simply by changing captions.

Whilst some would count this as anti-RAD, I would counter that it is actually better practice, leaving your project far more manageable and, from a personal standpoint, as a counter to BloatWare. The tab we are going to use in our next step is going to take on three stages, all on the same tab. Using a more traditional approach, we would end up with three or four separate dialogs to do the same as we are going to do with one. As dialogs are loaded either at start-up or from disk when required, my approach also leads to quicker applications, especially on slower machines.

Okay, so what are we going to do. Well we're going to collect information from the system about our user using Alan Mills' tutorial on getting System Metrics. (*Read his tutorial for all the information:)* We will then take this information and allow the user to press a button, and then add some more information of our own, give them and acceptance button, before finally providing them with the register option. All on one tab, and trust me, it won't be cluttered.

Bring TabSheet2 to the fore of our dialog. Name it Register, with the caption &Register. Next drop a TPanel on and set it's Align property to alClient and both Bevel properties to bvLowered. Finally remove the caption property by blanking the entry.

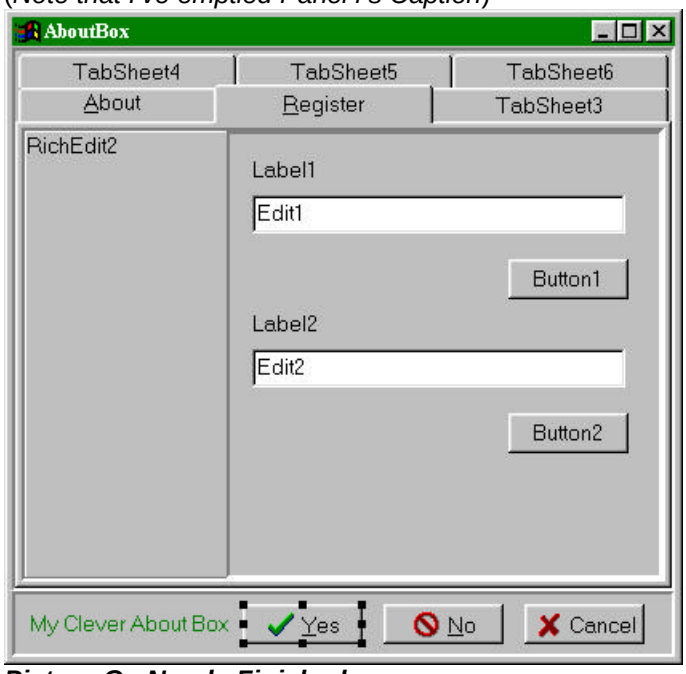
Now Add another two TPanels to our first, and set them up so it looks like this,

Picture F, The Start Of Our Register Tab

(*The left panel is with inner bevel = none, outer = raised, the right panel is both lowered*)

Next let's add some components, to the left panel add a TRichEdit, Name it InfoRich, and set it's other properties to
Align = alClient
Alignment = taLeftJustify
BorderStyle = bsNone
Color = clBtnFace
PlainText = false
ReadOnly = true
ScrollBars = ssVertical
TabStop = false
WordWrap = true

Now place 2 TEdit's, 2 TLabel's, and 2 TButtons onto the right panel, so it should now appear like this,
(*Note that I've emptied Panel4's Caption*)

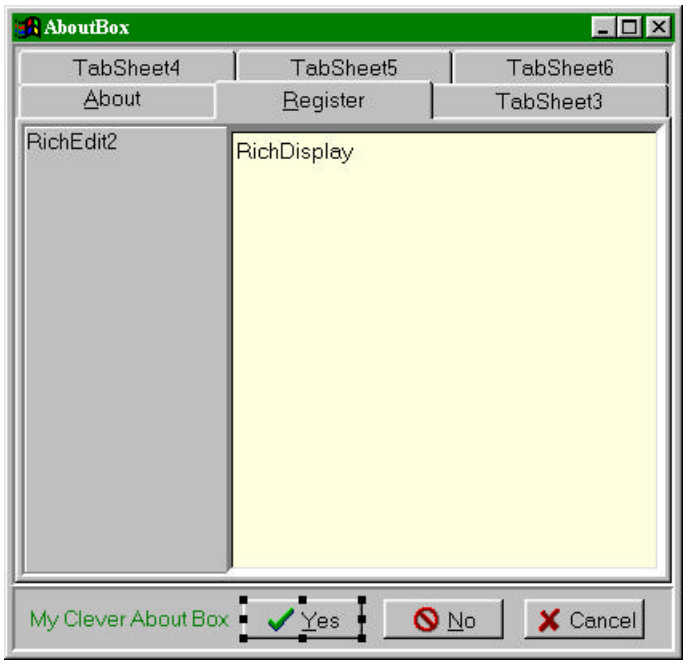


Picture G, Nearly Finished,

Finally, add a second TRichEdit to the right hand panel, and set it's properties as follows,

Align = alClient
Color = clInfoBk //You can choose your own if you wish
Name = RichDisplay
ReadOnly = true
ScrollBars = ssVertical
TabStop = false
Visible = false
WordWrap = true

Now our dialog looks like the following, at design time at least,



Picture H, Our finished box.

Run the application, select about from the menu, and then switch to the Register tab with the mouse, and you see the first dialog, show in Picture G. This is obviously because our RichEdit is invisible. Add the following code for now into Button1's Onclick handler,

(NB, change the caption to &Button1)

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
  RichDisplay->Visible = !RichDisplay->Visible;
}
```

Now run the application, bring up the about box, and select register. Click on button1, and our RichEdit appears, covering our buttons and editors. Press button1 again, which you can't do:). Okay, press Alt-B and you'll see the RichEdit disappear. Alt-R now you see it, Alt-R now you don't. Before closing the dialog, display the RichEdit, and then close the box. Now if you fire the dialog, the RichEdit is still visible, which we don't want, but we'll deal with that next.

What we are actually doing here is *running a window within our window*. That is we are going to control all the actions which take place in our dialog's register tab.

We can work with any component on a form in the same manner. Let's set up our Register tab so that it places all the components we want in the position we want, with the captions we want and finally the buttons we want on the bottom line.

Place the following code in the MasterPage OnChange handler event. (You'll have to select this from the drop down list in the object inspector as you'll be unable to select it on the form.)

```
void __fastcall TForm2::MasterPageChange(TObject *Sender)
{
  if(MasterPage->ActivePage->Name == "Register")
  {
    RichEdit2->Lines->Clear();
    RichEdit2->Lines->LoadFromFile("reg1.rtf");
    Button1->Visible = false;
    Button2->Visible = false;
    Edit1->ReadOnly = true;
```

```
    Edit1->Color = clInfoBk;
    Label1->Caption = "Your Name";
    Edit2->ReadOnly = true;
    Edit2->Color = clInfoBk;
    Label2->Caption = "Company Name";
    Edit1->Visible = true;
    Edit2->Visible = true;
    Butt2->Kind = bkAll;
    Butt2->Caption = "&Next";
    Butt2->Visible = true;
    Butt3->Kind = bkCancel;
    RichDisplay->Visible = false;
  }
}
```

Next run the application and type the following into our editor.

“

Welcome to the Registration Screen.

You are about to Register our product.

Check the details at right, if they're correct press next to continue, or close to exit.”

Note the leading blank lines, (You can copy the above and use Ctrl+V to paste into our editor if you wish.) Save this file as ref1.rtf in the default directory offered by the save box.

Next switch to the About box and the register tab. On the left you see the file you just typed, complete with font and colour settings. On the right you see our two captioned boxes, which we’ll fill in a moment, with the buttons we have chosen at the bottom.

To fill our user information I will use code lifted straight from Alan Mill’s excellent tutorial on getting the system metrics. If you want the following explained, read Alan’s tutorial. For now, close our app and add the following code to our OnChange handler.

```
void __fastcall TForm2::MasterPageChange(TObject *Sender)
{
  TRegistry *MyReg;
  AnsiString RegKey;
  OSVERSIONINFO vi;
  int PlatformNT;

  if(MasterPage->ActivePage->Name == "Register")
  {
    RichEdit2->Lines->Clear();
    RichEdit2->Lines->LoadFromFile("reg1.rtf");
    Button1->Visible = false;
    Button2->Visible = false;
    Edit1->ReadOnly = true;
    Edit1->Color = clInfoBk;
    Label1->Caption = "Your Name";
    Edit2->ReadOnly = true;
    Edit2->Color = clInfoBk;
    Label2->Caption = "Company Name";
    Edit1->Visible = true;
    Edit2->Visible = true;
    Butt2->Kind = bkAll;
    Butt2->Caption = "&Next";
    Butt2->Visible = true;
    Butt3->Kind = bkCancel;
    RichDisplay->Visible = false;

    //OS INFO
    vi.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
    GetVersionEx(&vi);

    switch (vi.dwPlatformId)
```

```
    {
      case VER_PLATFORM_WIN32s:
        PlatformNT = FALSE;
        break;
      case VER_PLATFORM_WIN32_WINDOWS:
        PlatformNT = FALSE;
        break;
      case VER_PLATFORM_WIN32_NT:
        PlatformNT = TRUE;
        break;
    }

    //REGISTRATION DETAILS
    if (PlatformNT)   //pat different for WindowsNT
    {
      RegKey = "\\Software\\Microsoft\\Windows NT\\CurrentVersion";
    }
    else
    {
      RegKey = "\\Software\\Microsoft\\Windows\\CurrentVersion";
    }

    MyReg = new TRegistry;  //create VCL component dynamically

    MyReg->RootKey = HKEY_LOCAL_MACHINE;
    if (MyReg->OpenKey(RegKey, FALSE))
    {
      Edit1->Text = MyReg->ReadString("RegisteredOwner");
      Edit2->Text = MyReg->ReadString("RegisteredOrganization");
    }

    MyReg->Free();  //remove registry component now we're done with it.
  }
}
```

and then add the following to the include section at the top,

```
#include <vcl\vcl.h>
#include <vcl\registry.hpp>
#pragma hdrstop
```

Now run the application, switch to the registry tab and you'll see the following, (obviously with different user details etc.)

Picture I, Our first stage Registration Screen

As you'll see from the above, we have not only chosen what to display, but changed most aspects of our designed screen. You could enable the editing of the Registry entries for name and company if you wished, but for our purposes we'll take them as they are. Not the style of the buttons in the bottom right hand corner, they are the standard Borland BitBtns, but we can change the caption and use them as our own.

Now of course, what happens when we click Next, we want to change these things. You could go through stages using the various components we have on our form, but for brevity, we'll assume that our next step is to present our user with the correct information, and let them choose finish.

This is where we'll get a little clever. The buttons Butt1-3, are available to every tab in our about box. To prove this, close the box, and reselect it. You should now have the initial about box, with the single close button on it. Select register, and we get both buttons. Select about again, and you still have both buttons.

If you wish to correct this, place the following in the MasterPage OnChange handler, which can be done either as an else to our current if, or as I prefer an if for each possible tab. The choice is yours.

```
if(MasterPage->ActivePage->Name = “About”)
{
  Butt2->Visible = false;
  Butt3->Kind = bkClose;
}
```

However, for demonstration purposes, I wish to leave the buttons visible at present.

In Butt2 OnClick handler enter the following code.

```
void __fastcall TForm2::Butt2Click(TObject *Sender)
{
  if(MasterPage->ActivePage->Name == "Register")
  {//we want to know about this page
    if(Butt2->Caption == "&Next")
    {//Capture the next press and create our finished file
      Butt2->Caption = "&Finish";
      RichDisplay->Lines->Clear();    //empty the display
      RichEdit2->Lines->LoadFromFile("reg2.rtf");
      RichDisplay->Lines->LoadFromFile("reg3.rtf");
      RichDisplay->Lines->Strings[6] = RichDisplay->Lines->Strings[6]+Edit1->Text;
      RichDisplay->Lines->Strings[7] = RichDisplay->Lines->Strings[7]+ Edit2->Text;
      RichDisplay->Visible = true;
      ModalResult = mrNone;
      return;
    }
    if(Butt2->Caption == "&Finish")
    {
      if(Application->MessageBox("About to Register Our Product",”Are You Sure",MB_YESNO
        |MB_ICONINFORMATION) == IDYES)
      {
        RichDisplay->Lines->SaveToFile("registered.rtf");
      }
    }
  }
}
```

Then create the two following text files,
reg2.rtf
“

Thank you for choosing to Register our product, press ***Finish*** and your registration will be complete.”

And reg3.rtf
“

Product Registerd = "Our Product Beta 0.2";

Registered To ::
Name ::
Company :: “

(*for ref3.rtf ensure that the seventh line is Name, and that line eight is company.*)

Now run the program, choose the register tab, click next, and finish, and presto:)

And Finally

To finish the Dialog box you should really handle the short coming in the TabSheet set up, that of the action of the Alt+underlined character. This requires handling in the same way as we did earlier for our Ctrl+T keypress, but this time within the Form2->KeyPreview set up. I’ll leave this, and other possibilities to your imagination as this tutorial is long already. *(Try using a TImage as a background to a tab, allowing edit boxes and buttons to appear disappear as required.)*

The only two things I’ll add is that the form can be set up in advance, unlike the calls to the built in dialogs I looked at in the first tutorial. You can call any tab to display first when you fire the box, a menu handler called register would launch straight into our menu tab with the following lines in its handler,

MasterPage->ActivePage = “Register”;
Form2->ShowModal;

You can also receive a result from the ShowModal dialog. You’ll notice in the last code example I set a variable ModalReturn to mrNone. I could have set it to any of the values defined in the BitBtn’s ModalReturn property, and then test the closing of our dialog as desired.

This has only been an introduction into what you can do with Dialog’s, remember they are just normal forms.

A Final Note

No liability is accepted by the author(s) for anything which may occur whilst following this tutorial