The Bits

From "The Bits" Website
http://www.cbuilder.dthomas.co.uk
emails regarding subject to : jon.jenkinson@mcmail.com

Dialog Boxes Part One

This is the first part of a two part series on working with Dialog Boxes. This tutorial takes you through the basics of using the standard dialog boxes built into Windows95 and C++ Builder themselves. The second tutorial, Dialog Boxes Part Two, will take you through building your own Dialogs, and show you the power you can attain quite easily withing Builder itself.

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.
```

Underlying Principals

The underlying principal behind a dialog is simple. A dialog is a window, or a form in BCB speak. As you can create a form, with any component(s) on it you like, so you can create a dialog, containing any component(s) you like.

In Part One of the series we will deal with the built in features of C++ Builder, the dialogs from the Dialogs tab, and MessageBox, the Builder interpretation of the primary Windows95 dialog.

There will be some quite self explanatory terms you will come across during this tutorial, which will be explained in line. The only part which may cause concern is sets. When required I will explain the rudimentary of what is going on with sets, but they provide a tutorial in themselves. Look elsewhere on the site if you need greater explanation.

To demonstrate the built in dialog types we will build a rudimentary text editor, used only to demonstrate the dialogs. Again, I will explain what is happening with the RTF component used for this editor, but if you wish greater depth, there is a tutorial at the site concerning itself purely with the RTF component.

Dialog Boxes, Part One, The Power Is Built In.

As mentioned in the introduction we are going to build a simple text editor. We will provide our editor with the following abilities, Open/Save, Font management, Colour, Print, Print setup, Find and Search/Replace.

The coding for these steps is minimal, the majority of the work being built into the dialogs we will load from the system. Spend at most two hours, and you'll have learnt how to use dialogs, and have a fully functional editor.

Follow the steps below, which are self explanatory.

Step One : Application->MessageBox, control.....

(a) Create an application form
(b) Hit run and smile :)

It doesn't do a lot yet does it. As mentioned in the introduction, you have effectively created a dialog box, that is, you have an application which interacts with the user. Still, that is not what we're here for, so let's actually build in some informative dialog boxes, and study the way that a standard Windows dialog can control your application.

Within the form's OnCreate function, add the following line,

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Application->MessageBox("Hello", "Within Form Create", MB_OK);
}
```

And run the application.

Now you do not see your form, you get a little dialog box, and it stops your application until you accept it. Do so now, and your form will appear. Close the form, and then put the following within the OnClose handler,

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
  Application->MessageBox("Bye Bye", "Within Form Close", MB_OK|MB_ICONERROR);
}
```

Now when you run the application, you get your first message, and now on closing we get another, complete with an icon. Similarly, again you program pauses until the box is accepted. This is because of the default attributes of the standard message box.

Let's look at the message box parameter list in more detail, before we move on.

Application->MessageBox(*char *Text, char *Caption, Unisgned Short Flags*);

The Text

The text is the message you wish to display, and is a standard “C” style char array. If you wish to pass an AnsiString type as your message, you must pass it via the AnsiString’s c_str() function, as follows,

```
AnsiString MessageToPass(“This is a test message”);
Application->MessageBox(MessageToPass.c_str(), “Test”, MB_OK);
```

The Caption

As with the text string above, this is a standard “C” style char array, which is displayed in the title bar of the message box.

**Important Note : If you wish to pass an empty string to the message box, you must pass it as a null string, that is “”, and not pass it a NULL. If you use the NULL, the application will compile, but the message box will not display. *Test this by changing the parameter in our call to FormClose, from “Bye Bye” to NULL. If you change the caption parameter to NULL, again it will compile, but this time you get a message box with the Caption = “Error”.*

**Another Important Note : The Text Parameter has no limit to its length, but will only wrap at 255 characters, resulting in a very wide box. The Caption length is limited to 255 characters. If you wish to retain control over the wrapping of the Text displayed, it is better to build your own dialog as we will do later.

The Flags

The flags parameter of the message box are the most important. They control how the message box performs when displayed, which buttons are displayed on the box, and which icon is displayed. The icon flag also controls which Windows System Sound is played when the box appears.

The next thing to note is that all message boxes are Windows system boxes, launched at the request of your application, and as such, Windows returns an integer value.

Display Button Flags

These are the flags used to control which buttons are displayed, and the return value returned.

Flag	Displays
MB_OK	A single push button, OK ***note: Default***
MB_OKCANCEL	2 push buttons, OK, Cancel, *default OK*
MB_YESNO	2 push buttons, Yes, No, *default Yes*
MB_RETRYCANCEL	2 push buttons, Retry, Cancel, *default Retry*
MB_ABORTRETRYIGNORE	3 push buttons, Abort, Retry, Ignore, *default Abort.*
MB_YESNOCANCEL	3 push buttons, Yes, No, Cancel, *default Yes*.
MB_HELP	Adds a Help button to the message box. Choosing the Help button or pressing F1 generates a Help event.

Return Values.	Define	Meaning

0		There was an error creating the message box.
1	IDOK	The OK button was used to close the box.
2	IDCANCEL	The Cancel button was used to close the box.
3	IDABORT	The Abort button was used to close the box.
4	IDRETRY	The Retry button was used to close the box.
5	IDIGNORE	The Ignore button was used to close the box.
6	IDYES	The Yes button was used to close the box.
7	IDNO	The No button was used to close the box.

Icon And Sound Control Flags

The following flags control the icon displayed in the message box, which in turn control the Windows System Sound played when the message box is displayed. Note the icon displayed is dependant on the version of the WinAPI on which your application is running.

Flag

MB_ICONEXCLAMATION *or* MB_ICONWARNING
displays *The exclamation mark*
MB_ICONINFORMATION *or* MB_ICONASTERISK
displays *The Information Icon, a lowercase i*
MB_ICONQUESTION
displays *A question mark icon*
MB_ICONSTOP *or* MB_ICONERROR *or* MB_ICONHAND
displays *The stop sign*.

Default Button Control

Controls which of the buttons displayed is the *highlighted* or default. If only one button is displayed, it is by default, the default button.

Flag	Meaning
MB_DEFBUTTON1	*Make the first button the default. Unless any of the following are given this is the default value*
MB_DEFBUTTON2	*Make the second button the default*
MB_DEFBUTTON3	*Make the third button the default*
MB_DEFBUTTON4	*Make the fourth button the default.*

Modality Flags

The modal flag affects how your message box is displayed. It also controls how the operating system, other applications, and your own application react to its display.

Flag

MB_APPLMODAL
The user must respond to the message displayed, and will be unable to access your application until they have done so. However, they will be able to move to other applications and continue their work there. If your application uses threads, the threads will continue execution, but all threads from the calling window, (*our application*), will not be able to accept user input until this box has been accepted. ***This is the default***.

MB_SYSTEMMODAL
The same as an MB_APPLMODAL, except that the box is displayed as the topmost window, that is in front of all other applications. Again, the user will not be able to access your application until this box has been cleared, but may continue to work with other applications.

MB_TASKMODAL
This will be little used, and basically is called if you wish to disable all top level windows in your application. As C++ Builder calls the message box from the top level application window, (invisible to you and I), this effectively happens anyway.

MB_DEFAULT_DESKTOP_ONLY
The desktop currently receiving input must be a default desktop; otherwise, the function fails. A default desktop is one an application runs on after the user has logged on. (*straight from the Help system, I do not know what they mean as I've not used Win95 as anything other than a single user system)*

MB_SETFOREGROUND
The message box is moved to become the foreground window. This is the default for all message boxes.

MB_TOPMOST
The message box is created with the WS_EX_TOPMOST window style. This stops other windows appearing over the top of the message box.

Text Display Control

The following flags are for controlling the way the text message is displayed.

Flag	Meaning
MB_RIGHT	The message is Right-Justified, (default left)
MB_RTLREADING	Displays the message and caption in right to left order

Windows NT Specific

MB_SERVICE_NOTIFICATION
Windows NT only: The caller is a service notifying the user of an event. The function displays a message box on the current active desktop, even if there is no user logged on to the computer.
If this flag is set, the hWnd parameter must be NULL. This is so the message box can appear on a desktop other than the desktop corresponding to the hWnd. For Windows NT version 4.0, the value of MB_SERVICE_NOTIFICATION has changed. See WINUSER.H for the old and new values. Windows NT 4.0 provides backward compatibility for pre-existing services by mapping the old value to the new value in the implementation of MessageBox and MessageBoxEx. This mapping is only done for executables that have a version number, as set by the linker, less than 4.0.

To build a service that uses MB_SERVICE_NOTIFICATION, and can run on both Windows NT 3.x and Windows NT 4.0, you have two choices.

1. At link-time, specify a version number less than 4.0; or
2. At link-time, specify version 4.0. At run-time, use the GetVersionEx function to check the system version. Then when running on Windows NT 3.x, use MB_SERVICE_NOTIFICATION_NT3X; and on Windows NT 4.0, use

MB_SERVICE_NOTIFICATION.

MB_SERVICE_NOTIFICATION_NT3X
Windows NT only: This value corresponds to the value defined for MB_SERVICE_NOTIFICATION for Windows NT version 3.51.

Step Two : A little More Control

As you can see from the list above, the simple Application->MessageBox is quite a powerful tool. When you are informing your users of something simple, it is most likely that you will use this type of box. To use any of the options, select one from the section you require and OR them together,
e.g.

Application->MessageBox(“Test”, “Test Message”,
MB_YESNO|MB_TOPMOST|MB_ICONERROR|MB_DEFBUTTON2);

will create a message box with a yes and no button, with the no button being default, an error icon/sound, and stay on top of other windows until accepted. Change the OnCreate handler of our form so it now looks like the following,

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
switch(Application->MessageBox("Test", "Test Message",
MB_YESNO|MB_TOPMOST|MB_ICONERROR|MB_DEFBUTTON2))
        {
        case IDYES:
                Form1->Caption = "Value Returned = IDYES";
                break;
        case IDNO:
                Form1->Caption = "Value Returned = IDNO";
                break;
        default:
                Form1->Caption = "Value Returned =Whoops";
                break;
        }
}
```

What you’ll now see is a little Yes/No box, with an Error icon/sound, and when you choose yes or no you set the forms Caption to the appropriate message. If you get the default message, you’ve done better than I, it should be impossible, and I cannot achieve it, but if anyone can, without cheating, let me know:)

Play with the options at your disposal, you will see later that we use the standard message box to help control our program and provide a standard *feel* to our application.

Step Three : The First of the Built In Dialogs.

Okay, we’ve dealt with the standard Windows message box, which is the dialog you will see a lot of. Now we’ll have a look at how to use the standard Dialogs provided with C++ Builder, (*note, as I only have access to the Standard Edition, I can only cover the ones available on my own version:(Also, this is a BIG section:)*)

First of all, clear the two event handlers be removing the code we typed in, (*leave the code in blue which was placed there by the IDE, the IDE will also remove it. Just remove the code we highlighted in Red, that is anything between the start { and end } of the function. If you wish you can simply do this by creating a new application.*

To demonstrate the dialogs we are going to build a very very basic text editor. We will use a TRichEdit as our editor, and all our dialogs will perform on this.

So, here we go. Drop a MainMenu on your form, from the standard tab, and name it Menu.

Double click on the Menu Icon, and you'll get a menu editor captioned Form1->Menu, with a highlighted box. Type &Dialogs in the caption property, and then name the menu Dialogs. Press return after naming it, and you get another highlighted box below it.

Type &Save in the caption property, and name it save, and then repeat for the following,

caption	name
&Load	Load
Fon&t	Font
&Colour	Colour
&Print	Print
Print Set&up	SetUp
&Find	Find
S&earch	Search

Okay, now run your application and make sure your menu displays correctly.

Now drop a TRichEdit on the form from the Win95 tab, and set its properties as follows,

Align	= alClient
Name	= REdit
ScrollBars	= ssVertical
PlainText	= false
WordWrap	= true

If you wish, and I do, set the Lines property to contain no lines by clicking the epsilon and deleting the word Redit.

Run the app again, and type some words into the RichEdit. Press Alt-F4 to terminate the application, and you lose your work. Let's implement the Save Dialog, two ways, one from our menu, and the other from the closing of our application.

From the Dialogs tab, drop a SaveDialog onto your form, (onto the REdit as it's set to alClient:)

set the properties of the dialog as follows

DefaultExt	= *.rtf		
Filter	= Click the epsilon and type	Rich Edit Text Files	*.rtf
		All Files	*.*
Name	= SaveBox		

double click the Options property, and set the following
ofOverwritePrompt = true

and finally set the Title to "Our Save Dialog."

Now to activate it. The RichEdit has a useful property called *Modified* and we'll use this to check if REdit has been modified, so we can see if there's anything to save. Place the following in the Save Menu's OnClick handler

```
void __fastcall TForm1::SaveClick(TObject *Sender)
{
  bool SaveYesNo;

  switch(REdit->Modified)
  {
    case false:
      switch(Application->MessageBox("Text hasn't Changed,  save anyway",
        "From within the menu Save OnClick", MB_YESNO|MB_ICONINFORMATION))
      {
        case IDYES:    //they want to save anyway
          SaveYesNo = true;
          break;
        case IDNO:     //they don't wish to save
        default:              //we'll assume they don't wish to save
          SaveYesNo = false;
          break;
      }//end switch
      break;
    case true:
      SaveYesNo = true;  //We're going to ask them
      break;
  }//end switch

  switch(SaveYesNo)
  {
    case true: //They Have something to save
      if(SaveBox->Execute())
      {  //They want to save
         REdit->Lines->SaveToFile(SaveBox->FileName);
      }
      else
      {
        //user changed their mind and chose not to save
      }
      break;
    case false:        //there's nothing for us to do
    default:           //let's ignore everything else
      break;
  }//end switch

  REdit->Modified = false;
}
```

Now, that looks quite complicated at first, but if you work through it, it's actually very straight forward. Firstly, recognise that the user has specifically requested a change, so if there's nothing to actually save, we should ask them if they really meant it. We do this in the first switch statement, if there's nothing changed since their last save, *modified* = *false*, we give them the option of confirming.

If something has changed, *modified* = *true*, then we assume they knew what they were doing. Now in the last switch, *switch(SaveYesNo)*, we know what the user wants. If our flag has been set to true, they want the save dialog, false, they don't, and we ignore it.

Then the save dialog itself. Simplest part of it really. The line, *if(SaveBox->Execute())*, takes advantage of the fact that if the cancel button is chosen within the dialog, it returns false. Assuming the user selects a valid filename, it is returned, including full path, as the property, *SaveBox->FileName*, which we use with the RichEdit->Lines->SaveToFile function to save our file.

Run the app, and do not type anything, just select the menu|Save option. To your surprise, you get the SaveBox dialog. This is because the RichEdit considers itself to have changed, a problem we'll overcome later when we look at the Load box.

For now, don't type anything in the dialog, just click the save button. You'll notice nothing happens, this is because the dialog requires a value in the FileName property, and wont return without one. Okay, click the cancel button, that is you don't want to save anything.

Finally we get to set the *modified* flag to false. Now, click the menu|Save again, and this time you'll be presented with our MessageBox, select No, and you return to your application, select Yes and you get the SaveBox.

Right, return to the application and type some text in, "Good morning" or something. Bring the save dialog up, and navigate to a directory of your choice, then type in "Test" as the filename, and click save.

Now bring the dialog up again, and notice two things. Firstly, the dialog has remembered where you last saved, and secondly it has added the .rtf extension for you. Double click test.rtf, and you get a system generated messagebox telling you the file already exists. Select no, and then right click in the test.rtf file.

You now get your standard Windows Explorer Action menu, with Select added to the top. To prove we have a simple RTF file, choose open from the pop up menu. You should now see WordPad open and display your file. (*if you have certain word processors intalled they override this association, the point being, it opens in whatever is associated with the rtf extension.*)

Close WordPad down, and pick "All Files" from the "Save As Type" drop down box in our save dialog. If you have any other files in the directory, you can open them, copy them, move them, etc etc., in fact you can do anything you could do with the normal Win95 API save dialog, (including launching other applications:)

Next we'll improve our text editor by catching the OnClose event of the form, and check to see if the text has changed again. We could override the Save:OnClick handler to accept another parameter, but for now we'll do the more formal method, which allows the user to cancel and return to the editor

In the OnCloseQuery event, place the following

```
void __fastcall TForm1::FormCloseQuery(TObject *Sender, bool &CanClose)
{
  if(REdit->Modified)
  {
  switch(Application->MessageBox("Save Changes to Text?","Our Editor", MB_YESNOCANCEL
    | MB_ICONWARNING))
  {
    case IDYES:
      if(SaveBox->Execute())
      {//They want to save
        REdit->Lines->SaveToFile(SaveBox->FileName);
      }
      else
      {//they changed their mind,  assume they want the editor back
        CanClose = false;
      }
      break;
    case IDNO:         //Ignore it they don't want to save
      break;
    case IDCANCEL:            //They wish to return to the editor
      CanClose = false;
      break;
    default:           //we shouldn't get here,  but just in case
      throw Exception("Whoops, you've reached a place your not supposed to get to");
      break;
    }//end switch
  }//end if REdit
}
```

(editor's note: If you don't understand exceptions, there are two articles on exceptions in our object oriented programming section).

Now when you run the application, simply close the form. Rather than using the OnClose handler, we perform an OnCloseQuery. This allows us to ask the user for an opinion, that is do they wish to save or not, and if they choose cancel from either our message box or our save dialog, we can return them to our editor by setting the CanClose boolean to false.

Now, before we get to our load dialog, let's get rid of this Modified property, unless we actually type in some text.

In the forms OnShow handler put the following,

```
void __fastcall TForm1::FormShow(TObject *Sender)
{
  REdit->Modified = !REdit->Modified;
}
```

Now when you run the application you'll find you can exit without any annoying dialogs, but beware that if you action the OnHide event handler of the form, you must look after the *Modified* flag as well. So for completeness, place the following in the OnHide handler,

```
void __fastcall TForm1::FormHide(TObject *Sender)
{
  REdit->Modified = !REdit->Modified;
}
```

The OnHide is something you call explicitly, take my word that the above switches the value of the modified flag. The logic, if it's true, and we hide it, we switch it to false. Next, when we show the form, we switch it back to true. (*This is a simple work around, and will not handle if your form closes when it is hidden:(*

<u>**Step Four : The Load Dialog**</u>

Onto our next Dialog. From our menu we can see that it is the load option. Before we actually implement it, let's give ourselves something to load. Run the application and type some lines of text into the richedit and save it with a name that suits you. *To prove it's an RTF file, copy this and the previous two paragraphs, taking in the coloured and typefaced words. Use ctrl-V to paste into our editor.*

Okay, onto the load dialog. Drop an Open Dialog onto the form from the Dialogs tab. Set the properties as follows

DefaultExt	= *.rtf
Filter	= Rich Text Edit Files|*.rtf|All Files|*.*
Name	= OpenBox
Options	
ofFileMustExist	= true
Title	= Our Load Box

Now place the following in the menu|Load onclick handler,

```
void __fastcall TForm1::LoadClick(TObject *Sender)
{
  SaveClick(Sender);
  if(OpenBox->Execute())
  {
    REdit->Lines->LoadFromFile(OpenBox->FileName);
  }
  REdit->Modified = false;
}
```

Before you run the application, check what we're doing. If the user hasn't typed any text, we're going to tell them this with the call to the SaveClick function, if you remember how we wrote that. This is not ideal, and really, we should write a proper save check here, but for clarity, I've simply called the function.

Next we show the Open dialog, and if you select a file, we load it in. Select the file we just saved, and note that it comes back with any colours and formatting we saved it with. We set the *modified* flag to stop the save if we exit or save without changing anything.

That's it. So far, we have used three very powerful dialog boxes, without too much code. For those who remember the old days of file handling under DOS or Windows, notice how easy it has been to load and save a file under C++ Builder. Similarly, notice how easy it has been to get the operating system feel without any complex coding.

**Step Five	: The Font Dialog:(**

Ah, I spoke to soon. The font dialog requires us to use sets, and involves a little more code to provide a useful dialog. However, once you've done this once, you can apply the code here to most uses of this dialog that you'll come across.

From the Dialogs tab, drop a FontDialog onto the form, and set the properties as follows

```
Name                    = FontBox
Options
        fdEffects       = true
        fdForceFontExist        = true
```

Next, in the menu|font OnClick handler, type the following,

```
void __fastcall TForm1::FontClick(TObject *Sender)
{
  FontBox->Execute();
}
```

Note, we are not really interested in whether the user accepts or rejects the dialog. This is because the font dialog comes with an event handler, unlike the ones we've looked at so far.

A font dialog has an event handler called OnApply. For now, run the application and select the font from the menu. The dialog appears, with all the options you're used to seeing. Note that it starts with MS Sans Serif as the chosen font, and 8 as the size and black as the colour etc. This is because we haven't changed the Font properties of the dialog box.

In a text editor, we would like our font box to launch with the properties at the cursor. We do this by adding code just prior to our call to the FontBox.

```
void __fastcall TForm1::FontClick(TObject *Sender)
{

  TFontStyles EmptySet;
  int Start, Length;

  FontBox->Font->Color = REdit->SelAttributes->Color;
  FontBox->Font->Height = REdit->SelAttributes->Height;
  FontBox->Font->Name = REdit->SelAttributes->Name;
  FontBox->Font->Pitch = REdit->SelAttributes->Pitch;
  FontBox->Font->Size = REdit->SelAttributes->Size;
  FontBox->Font->Style = EmptySet;

  if(REdit->SelAttributes->Style.Contains(fsBold))
  {
    FontBox->Font->Style = FontBox->Font->Style << fsBold;
  }
  if(REdit->SelAttributes->Style.Contains(fsItalic))
  {
    FontBox->Font->Style = FontBox->Font->Style << fsItalic;
  }

  if(REdit->SelAttributes->Style.Contains(fsUnderline))
  {
    FontBox->Font->Style = FontBox->Font->Style << fsUnderline;
  }

  if(REdit->SelAttributes->Style.Contains(fsStrikeOut))
  {
    FontBox->Font->Style = FontBox->Font->Style << fsStrikeOut;
  }

  FontBox->Execute();
}
```

This uses the RichEdit SelAttribute property, and the Font and SelAttibute properties, whilst compatible, are not of the same type, so we can't use a simple =. If we were to use the Font property of RichEdit, we would not pick up the attributes at the cursor. (*I apologise about the complex use of sets here, it is important to understand these, but in the part two, when we write our own dialog, I explain this in detail. Suffice to say this works, but because of the way the Dialogs are kept in memory, we have to go through the longwinded process above, unless you know different:)*

If you took my hint before, you'll have an RTF file saved with colour formatting and a different font from the default.

Prove this now, by running the program, and select font from the menu before loading anything. You should see a particular font name, size, and colour, dependant on your version of RichEdit properties. Mine throws up MS Sans Serif, 8, Black. Press OK or Cancel, and load in the RTF file we saved earlier. If you copied my text, this time when you launch the FontBox you'll see Times New Roman, 10 and Navy. Hit OK or Cancel and close the application.

Now we'll use the FontBox OnApply event handler to apply any changes to the font settings we choose to make. In the handler, type the following,

```
void __fastcall TForm1::FontBoxApply(TObject *Sender, HWND Wnd)
{
  TFontStyles EmptySet;
  int Start, Length;

  REdit->SelAttributes->Color = FontBox->Font->Color;
  REdit->SelAttributes->Height = FontBox->Font->Height;
  REdit->SelAttributes->Name = FontBox->Font->Name;
  REdit->SelAttributes->Pitch = FontBox->Font->Pitch;
  REdit->SelAttributes->Size = FontBox->Font->Size;
  REdit->SelAttributes->Style = EmptySet;

  if(FontBox->Font->Style.Contains(fsBold))
  {
    REdit->SelAttributes->Style = REdit->SelAttributes->Style << fsBold;
  }

  if(FontBox->Font->Style.Contains(fsItalic))
  {
    REdit->SelAttributes->Style = REdit->SelAttributes->Style << fsItalic;
  }

  if(FontBox->Font->Style.Contains(fsUnderline))
  {
    REdit->SelAttributes->Style = REdit->SelAttributes->Style << fsUnderline;
  }

  if(FontBox->Font->Style.Contains(fsStrikeOut))
  {
    REdit->SelAttributes->Style = REdit->SelAttributes->Style << fsStrikeOut;
  }

  Start = REdit->SelStart;
  Length = REdit->SelLength;
  REdit->SelText = REdit->SelText;
  REdit->SelStart = Start;
  REdit->SelLength = Length;
}
```

Now run the application, and without loading anything, change the font settings, try a different font, size and colour. Click the apply button, and then close the dialog with either OK or Cancel. Type in some text and it now has the attributes you just selected.

There is one problem here, that when you hit the apply button, the dialog doesn't close. For clarity earlier I left out an important feature, which we will now add. Change the menu|font click handler to the following,

```
void __fastcall TForm1::FontClick(TObject *Sender)
{

  TFontStyles EmptySet;
  int Start, Length;

  FontBox->Font->Color = REdit->SelAttributes->Color;
  FontBox->Font->Height = REdit->SelAttributes->Height;
  FontBox->Font->Name = REdit->SelAttributes->Name;
  FontBox->Font->Pitch = REdit->SelAttributes->Pitch;
  FontBox->Font->Size = REdit->SelAttributes->Size;
  FontBox->Font->Style = EmptySet;

  if(REdit->SelAttributes->Style.Contains(fsBold))
  {
    FontBox->Font->Style = FontBox->Font->Style << fsBold;
  }

  if(REdit->SelAttributes->Style.Contains(fsItalic))
  {
    FontBox->Font->Style = FontBox->Font->Style << fsItalic;
  }

  if(REdit->SelAttributes->Style.Contains(fsUnderline))
  {
    FontBox->Font->Style = FontBox->Font->Style << fsUnderline;
  }

  if(REdit->SelAttributes->Style.Contains(fsStrikeOut))
  {
    FontBox->Font->Style = FontBox->Font->Style << fsStrikeOut;
  }

  if(FontBox->Execute())
  {
    REdit->SelAttributes->Color = FontBox->Font->Color;
    REdit->SelAttributes->Height = FontBox->Font->Height;
    REdit->SelAttributes->Name = FontBox->Font->Name;
    REdit->SelAttributes->Pitch = FontBox->Font->Pitch;
    REdit->SelAttributes->Size = FontBox->Font->Size;
    REdit->SelAttributes->Style = EmptySet;

    if(FontBox->Font->Style.Contains(fsBold))
    {
      REdit->SelAttributes->Style = REdit->SelAttributes->Style << fsBold;
    }

    if(FontBox->Font->Style.Contains(fsItalic))
    {
      REdit->SelAttributes->Style = REdit->SelAttributes->Style << fsItalic;
    }

    if(FontBox->Font->Style.Contains(fsUnderline))
    {
      REdit->SelAttributes->Style = REdit->SelAttributes->Style << fsUnderline;
    }

    if(FontBox->Font->Style.Contains(fsStrikeOut))
    {
      REdit->SelAttributes->Style = REdit->SelAttributes->Style << fsStrikeOut;
    }

    Start = REdit->SelStart;
    Length = REdit->SelLength;
    REdit->SelText = REdit->SelText;
    REdit->SelStart = Start;
    REdit->SelLength = Length;
  }
}
```

Now then, we have quite a bit of functionality built into our editor. Run the application, and without loading anything, bring up the font box. Change the font style, colour, size etc, and press the okay button. Now you see the new font settings.

Highlight this text using either the mouse or press <shift>+<home>. Once your text is highlighted, bring up the font dialog again, and change the settings. Move the dialog so you can see your highlighted text and press the Apply button, your text changes instantly:)

Now close the dialog with either OK or Cancel, *the changes have already been applied*, and move the cursor down a line using the down arrow. Type some more text and notice that it appears in the original font settings. That is, the only pieces of your text to change are the ones that you've elected to change. This is the same as most Word processors on the market. For another time, you can change these default font options of the RichEdit, as it would be sensible to add a status bar or picklist to display which font has focus etc. As I say, in the interest of brevity, I'll leave those for a RichEdit tutorial.

Step Six : The Colour Dialog

Let's move to the next Dialog in our list, the Colour Dialog. We'll use this to change just the colour of our text. Drop a TColorDialog on the form and name it ColourBox.

In the Menu|Colour on click handler place the following code,

```
void __fastcall TForm1::ColourClick(TObject *Sender)
{
  ColourBox->Color = REdit->SelAttributes->Color;

  if(ColourBox->Execute())
  {
    REdit->SelAttributes->Color = ColourBox->Color;
    REdit->SelText = REdit->SelText;
  }
}
```

Okay, run the app and type in some text. Highlight the text and fire the Colour dialog from the menu. When the dialog opens, you will see the standard Windows Color Dialog. You can select any of the pre-defined colours and click OK, your text changes to the selected colour. Providing you stay within the pre-defined colours, if you check the colour with the FontBox, you'll see the colour displayed their.

However, if you define your own colour, and apply it to the text, you will find that, whilst it changes the colour of your text, it wont be displayed in the FontBox, in fact, it confuses all the attributes of the FontBox. Simple problem, the standard Font dialog expects a pre-defined colour, and we're sending it one it doesn't know about. This is a small problem, but you should note it. Further, the RichEdit is quite happy with the user defined colour, and will save and load the attributes as you define them.

Step Seven : The Print Dialog

Next dialog is the Print, and the Print SetUp. Place one of each on your form, the PrintDialog named ToPrinter, and the PrinterSetupDialog named PrintSetUp. Set up the two menu handlers as follows

```
void __fastcall TForm1::PrintClick(TObject *Sender)
{
  String Caption("Draft RichEdit Print");

  if(ToPrinter->Execute())
  {
    REdit->Print(Caption);
  }
}

void __fastcall TForm1::SetUpClick(TObject *Sender)
{
  PrintSetUp->Execute();
}
```

This will print the document via Windows standard functions. The RichEdit allows you to control the print better by using its PageRect property to set dimensions, but that is beyond the scope of this tutorial. The options available to the Print Dialogs are listed in the Object Inspector, and are simply properties accessed in the normal way.

To deal with the printer set up options etc is a very long tutorial beyond the scope here. Surfice, if you follow the code above, your text editor will print as well as Notepad, including all WYSIWYG functions such as fonts colours etc. You will not get pages unless you set up the PageRect property which involves calculating in inches/cm/pixels and something called wips?

Step Eight : The Find Dialog

Onto the find dialog. This is a little more complex to implement, but here goes. Drop a FindDialog on to the form and set its properties as follows,

Name = FindBox
Options
 frDown = true
 frFindNext = true
 frHideUpDown = true

Now add the two following handlers via the Object Inspector

```
void __fastcall TForm1::FindClick(TObject *Sender)
{
  FindBox->Execute();
}
```

Simply execute the box. Note that the box stays active until it is cancelled by the user.

```
void __fastcall TForm1::FindBoxFind(TObject *Sender)
{
  TSearchTypes SearchSet;
  int StartFound;

  if(REdit->SelStart != 0)
  {
    REdit->SelStart = REdit->SelStart + REdit->SelLength;
  }
  if(FindBox->Options.Contains(frWholeWord))
  {
    SearchSet << stWholeWord;
  }
  else
  {
    SearchSet >> stWholeWord;
  }

  if(FindBox->Options.Contains(frMatchCase))
  {
    SearchSet << stMatchCase;
  }
  else
  {
    SearchSet >> stMatchCase;
  }

  StartFound = REdit->FindText(FindBox->FindText,  REdit->SelStart,
    REdit->Lines-  >Text.Length(),SearchSet);

  REdit->SelStart = StartFound;
  REdit->SelLength = FindBox->FindText.Length();
  REdit->SetFocus();

  if(REdit->SelStart == -1)
  {
    Application->MessageBox("String Not Found", "FindBox",MB_OK|MB_ICONERROR);
  }
 }
```

To explain. When the user clicks the find next button, they fire a Find event, (*FindBoxFind*), and we set about handling it. TRichEdit has a method called FindText, and this is what we use to do the search. The main thing to note here is the use of sets to build up SearchSet, the attributes which the function does.

Surfice to say that the thing works, run the application and check out how the Find function works. *The REdit->SetFocus() call is to provide visual feedback to the user. For some reason, it seems that the FindDialog box overrides the normal display properties of the system ??? Oh well:)*

<u>**Step Nine : The Find/Replace Dialog.**</u>

Finally, the search and replace dialog. This does become a little more difficult, not in the actual find and replace, but because you have an option of Replace All. To start with, let's drop a ReplaceDialog on the form and set its properties as follows,

Name = ReplaceBox
options
    frDown = true
    frFindNext = true
    frHideUpDown = true
    frReplace = true

Then place the following handlers,

Call and fire the Box,

```
void __fastcall TForm1::SearchClick(TObject *Sender)
{
  ReplaceBox->Execute();
}
```

Then place the following in the find handler, (*this is the same as our FindBox find, and again we* *could have a function to do both in one, but that would be beyond our scope for now:)*

```
void __fastcall TForm1::ReplaceBoxFind(TObject *Sender)
{
TSearchTypes SearchSet;
int StartFound;

if(REdit->SelStart != 0)
        {
        REdit->SelStart = REdit->SelStart + REdit->SelLength;
        }

if(ReplaceBox->Options.Contains(frWholeWord))
        {
        SearchSet << stWholeWord;
        }
else
        {
        SearchSet >> stWholeWord;
        }

if(ReplaceBox->Options.Contains(frMatchCase))
        {
        SearchSet << stMatchCase;
        }
else
        {
        SearchSet >> stMatchCase;
        }

StartFound = REdit->FindText(ReplaceBox->FindText,
                              REdit->SelStart,
                              REdit->Lines->Text.Length(),
                                      SearchSet);

REdit->SelStart = StartFound;
REdit->SelLength = ReplaceBox->FindText.Length();
REdit->SetFocus();

if(REdit->SelStart == -1)
        {
        Application->MessageBox("String Not Found","FindBox",
                              MB_OK|MB_ICONERROR);
        }
}
```

Next we'll implement a handler for the Replace, *And*, the ReplaceAll. Type the following into the OnReplace handler,

```
void __fastcall TForm1::ReplaceBoxReplace(TObject *Sender)
{
  int temp;
  TSearchTypes SearchSet;
  int Found;
  bool ReplaceAllFlag = false;

  if(ReplaceBox->Options.Contains(frWholeWord))
  {
    SearchSet << stWholeWord;
  }
  else
  {
    SearchSet >> stWholeWord;
  }

  if(ReplaceBox->Options.Contains(frMatchCase))
  {
    SearchSet << stMatchCase;
  }
  else
  {
    SearchSet >> stMatchCase;
  }

  if(ReplaceBox->Options.Contains(frReplaceAll))
  {
    ReplaceAllFlag = true;
  }

  do
  {
    Found = REdit->FindText(ReplaceBox->FindText, REdit->SelStart,
        REdit->Lines->Text.Length(),SearchSet);

    Form1->Caption = Found;
    if(Found == -1)
    {
      Application->MessageBox("All Occurances Replaced","ReplaceBox",MB_OK|MB_ICONERROR);
      ReplaceAllFlag = false;
    }
    else
    {
      REdit->SelStart = Found;
      REdit->SelLength = ReplaceBox->ReplaceText.Length();
      temp = REdit->SelStart;
      REdit->SelText = ReplaceBox->ReplaceText;
      REdit->SelStart = temp;
      REdit->SelLength = ReplaceBox->ReplaceText.Length();
      REdit->SetFocus();
    }
  } while(ReplaceAllFlag == true);
}
```

Run the application and test it, first the Replace function and then the ReplaceAll. Work through the code slowly, again paying attention to the use of sets, in this case to set flags to allow for a multiple search and replace operation. This could be coded a little tighter, but for clarity I've layed it out in the above manner so you can see more easily the process.

Before moving on, ensure you are happy with the way the standard dialogs and message boxes work before you go on to create your own dialog by reading the next in the series, Dialog Boxes, Build Your Own !

A Final Note

##~Warn them of anything that is extra important, e.g.
As with all DLL's, you need to know the functions available, and their parameter list before you can successfully call them.

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