TRichEdits And Embedded Objects

by Brian Long

The TRichEdit component is a representation of the Win32 rich text control, version 1.0. Dave Jewell wrote a pair of articles in Issues 34 and 35 which discussed how to surface functionality from the version 2.0 rich text control into a Delphi component. This article discusses how to make use of functionality that exists in the version 1 control, but which the VCL does not make directly available to you. To start us off, here is a question that arrived at *The Delphi Clinic* a few weeks ago:

'When I create a document with WordPad and save it as an RTF file, I can open it into a TRichEdit *and view the contents. If I tell WordPad to insert a bitmap into the document (using* Insert | Object...*, or even copy and paste) I can see that the file contains the picture object by loading it back into WordPad, or by using a DOS text viewer. However, if the file is opened back into a* TRichEdit*, it happily displays the text, but not the picture. How do I display the picture? Or on a more generic scope, how do I alter the* TRichEdit *component so that other objects can be interpreted?'*

Win32 RichEdit controls, as encapsulated by TRichEdit components, by default do not support OLE objects. When you choose Insert | Object... in WordPad, you are asking it to embed an OLE object into the document. In order to entice your TRichEdit into supporting them, you must implement an IRichEditOleCallback interface in some arbitrary object, and tell the TRichEdit about it with an em_SetOleCallback message. This message is not in the Win32 API help file that comes with Delphi (even Delphi 5), but it is described in the MSDN Library. Once this has been done, you will be able to paste and insert objects into your TRichEdit, and also edit them by double clicking.

The IRichEditOleCallback interface is shown in Listing 1. In my representation of the interface, some of the methods are marked as safecall and some as stdcall. The ones not marked as safecall need very specific HResult codes returned under some circumstances, controlled by code. If the implications of safecall are unclear, see my explanatory article on the safecall reserved word in last month's issue.

Whilst the job of implementing this interface may look a little daunting, many of the methods can be stubbed out, and some of them can have their implementation effectively copied from appropriate bits of the VCL. Listing 2 has the code that does the job, which will be explained in the remainder of this article.

➤ *Listing 1*

```
IRichEditOleCallback = interface(IUnknown)
  ['{00020D03-0000-0000-C000-000000000046}']
  function GetNewStorage: IStorage; safecall;
  procedure GetInPlaceContext(out Frame: IOleInPlaceFrame;
    out Doc: IOleInPlaceUIWindow; var FrameInfo: TOleInPlaceFrameInfo); safecall;
  procedure ShowContainerUI(fShow: Bool); safecall;
  procedure QueryInsertObject(const ClsID: TCLSID;
    Stg: IStorage; CP: Longint); safecall;
  procedure DeleteObject(OleObj: IOleObject); safecall;
  procedure QueryAcceptData(dataobj: IDataObject; var cfFormat: TClipFormat;
    reCO: DWord; fReally: Bool; hMetaPict: HGlobal); safecall;
  function ContextSensitiveHelp(fEnterMode: Bool): HResult; stdcall;
  function GetClipboardData(const ChRg: TCharRange; reCO: DWord;
    out DataObj: IDataObject): HResult; stdcall;
  procedure GetDragDropEffect(fDrag: Bool; grfKeyState: DWord;
    var dwEffect: DWord); safecall;
  procedure GetContextMenu(SelType: Word; OleObj: IOleObject;
    const ChRg: TCharRange; var Menu: HMenu); safecall;
end;
```

The class that implements this interface in this case is called TRichEditOleCallback. This class inherits from TInterfacedObject to avoid me having to worry about the three methods from IUnknown that also need to be implemented. The class has a constructor that simply records which TRichEdit it is associated with, and a destructor that does some tidying up.

To start with, we can dispense with ContextSensitiveHelp and GetClipboardData, as this class does not implement these features. So they both return an HResult code of E_NOTIMPL. Also, GetContextMenu does nothing more than set its Menu parameter to 0 and GetDragDropEffect does nothing at all (the passed in dragging operation is probably just fine, so we leave it). Likewise QueryAcceptData and QueryInsertObject do nothing, indicating that we are happy to have any OLE object pasted or inserted.

This does not leave too many methods now. DeleteObject is called when the OLE object is to be deleted from the rich edit control. We deal with this by calling the IOleObject parameter's Close method with a parameter of OLECLOSE_NOSAVE.

The job of GetNewStorage is to allocate storage for the OLE object, in the shape of an IStorage interface. In this case, the rich edit is acting as a container for an OLE object. There is already a TOleContainer class in the VCL that does a very similar job, so when we wonder what to do, it's a good idea to look at the source for it. In fact it has a method called CreateStorage that does just what we need, so I pinched its calls to the weirdly named CreateILockBytesOnHGlobal and StgCreateDocfileOnILockBytes APIs.

Now we have GetInPlaceContext and ShowContainerUI to deal with. The former of these two has a requirement to provide an IOleInPlaceFrame interface reference that represents the underlying form housing the rich edit. It also needs to fill in a TOleInPlaceFrameInfo record with information about the window,

```
TOleRichEdit = class(TRichEdit)
protected
  procedure CreateHandle; override;
end;
TRichEditOleCallback =
  class(TInterfacedObject, IRichEditOleCallback)
private
  FOwner: TRichEdit;
protected
  { IRichEditOleCallback }
  function GetNewStorage: IStorage; safecall;
  procedure GetInPlaceContext(out Frame: IOleInPlaceFrame;
    out Doc: IOleInPlaceUIWindow; var FrameInfo:
    TOleInPlaceFrameInfo); safecall;
  procedure ShowContainerUI(fShow: Bool); safecall;
  procedure QueryInsertObject(const ClsID: TCLSID; Stg:
    IStorage; CP: Longint); safecall;
  procedure DeleteObject(OleObj: IOleObject); safecall;
  procedure QueryAcceptData(dataobj: IDataObject;
    var cfFormat: TClipFormat; reCO: DWord; fReally: Bool;
    hMetaPict: HGlobal); safecall;
  function ContextSensitiveHelp(fEnterMode: Bool): HResult;
    stdcall;
  function GetClipboardData(const ChRg: TCharRange; reCO:
    DWord; out DataObj: IDataObject): HResult; stdcall;
  procedure GetDragDropEffect(fDrag: Bool; grfKeyState:
    DWord; var dwEffect: DWord); safecall;
  procedure GetContextMenu(SelType: Word; OleObj:
    IOleObject; const ChRg: TCharRange; var Menu: HMenu);
    safecall;
public
  constructor Create(Owner: TRichEdit);
  destructor Destroy; override;
end;
{ TRichEditOleCallback }
constructor TRichEditOleCallback.Create(Owner: TRichEdit);
begin
  inherited Create;
  FOwner := Owner
end;
destructor TRichEditOleCallback.Destroy;
var Form: TCustomForm;
begin
  Form := GetParentForm(FOwner);
  if Assigned(Form) and Assigned(Form.OleFormObject) then
    (Form.OleFormObject as
    IOleInPlaceUIWindow).SetActiveObject(nil, nil);
  inherited;
end;
function TRichEditOleCallback.ContextSensitiveHelp(
  fEnterMode: Bool): HResult;
begin
  Result := E_NOTIMPL
end;
procedure TRichEditOleCallback.DeleteObject(OleObj:
  IOleObject);
begin
  OleObj.Close(OLECLOSE_NOSAVE)
end;
function TRichEditOleCallback.GetClipboardData(
  const ChRg: TCharRange; reCO: DWord;
  out DataObj: IDataObject): HResult;
begin
  Result := E_NOTIMPL
end;
procedure TRichEditOleCallback.GetContextMenu(
  SelType: Word; OleObj: IOleObject;
  const ChRg: TCharRange; var Menu: HMenu);
begin
  Menu := 0
end;
procedure TRichEditOleCallback.GetDragDropEffect(fDrag:
  Bool; grfKeyState: DWord; var dwEffect: DWord);
begin
  //Use normal effect (stored in dwEffect)
end;
procedure TRichEditOleCallback.GetInPlaceContext(
  out Frame: IOleInPlaceFrame; out Doc: IOleInPlaceUIWindow;
  var FrameInfo: TOleInPlaceFrameInfo);
var Form: TCustomForm;
begin
  //Get richedit's underlying form
  Form := ValidParentForm(FOwner);
  //Ensure there is a TOleForm object
  if Form.OleFormObject = nil then
    TOleForm.Create(Form);
  //Get relevant frame interface
  Frame := Form.OleFormObject as IOleInPlaceFrame;
  Doc := nil; //Document window is same as frame window
  FrameInfo.hWndFrame := Form.Handle;
  FrameInfo.fMDIApp := False;
  FrameInfo.hAccel := 0;
  FrameInfo.cAccelEntries := 0;
end;
function TRichEditOleCallback.GetNewStorage: IStorage;
var
  LockBytes: ILockBytes;
begin
  //Basically copied from TOleContainer.CreateStorage
  OleCheck(CreateILockBytesOnHGlobal(0, True, LockBytes));
  OleCheck(StgCreateDocfileOnILockBytes(LockBytes,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE or STGM_CREATE,
      0, Result));
end;
procedure TRichEditOleCallback.QueryAcceptData(dataobj:
  IDataObject; var cfFormat: TClipFormat; reCO: DWord;
  fReally: Bool; hMetaPict: HGlobal);
begin
  //Accept anything
end;
procedure TRichEditOleCallback.QueryInsertObject(const
  ClsID: TCLSID; Stg: IStorage; CP: Integer);
begin
  //Accept anything
end;
procedure TRichEditOleCallback.ShowContainerUI(fShow: Bool);
var
  Form: TCustomForm;
begin
  if fShow then begin
    Form := GetParentForm(FOwner);
    if Assigned(Form) and Assigned(Form.Menu) then begin
      //Disassociate OLE menu handle from UI menu
      Form.Menu.SetOle2MenuHandle(0);
      //Make sure any space that was made for in-place
      //toolbars is reclaimed
      (Form.OleFormObject as IVCLFrameForm).ClearBorderSpace
    end
  end
end;
{ TOleRichEdit }
procedure TOleRichEdit.CreateHandle;
begin
  inherited;
  Perform(em_SetOleCallback, 0,
    Longint(TRichEditOleCallback.Create(Self) as
      IRichEditOleCallback))
end;
```

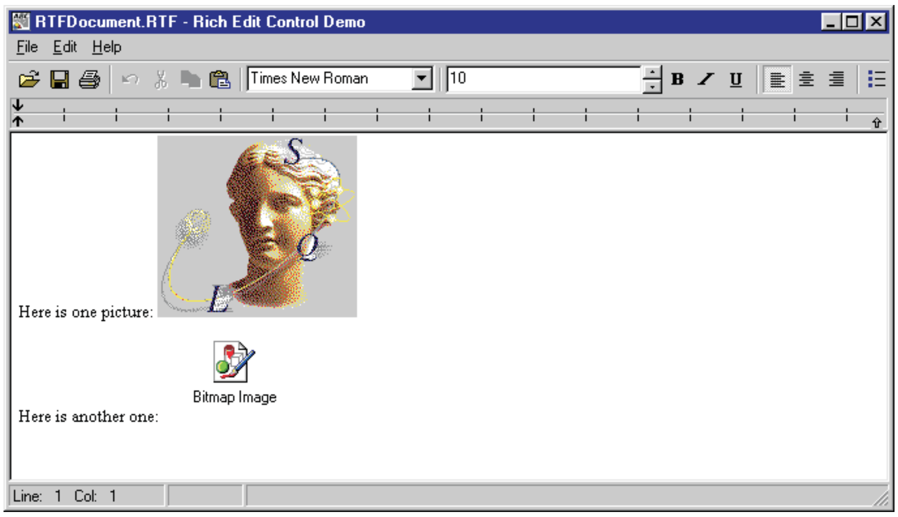
➤ *Listing 2*

including the window handle, whether it is an MDI app, and so on. This information is used when the object is being prepared for in-place editing, so the underlying application that 'owns' the object can insert its menus and generally take over the user interface.

How do we get one of the `IOleInPlaceFrame` interfaces? Well, we go back to the `OleCtnrs` unit for this. The interface section of the unit defines an interface called `IVCLFrameForm` which is based on `IOleInPlaceFrame`. In the unit's implementation section, there is a function `GetVCLFrameForm` that extracts an `IVCLFrameForm` out of a given form using the form's `OleFormObject` property. This interface property may be `nil`, so `GetVCLFrameForm` checks and, if so, assigns a freshly created `TOleForm` to it. Since the property is an interface reference, we do not need to worry about destroying this object. It will be automatically destroyed through reference counting.

So the implementation of `GetInPlaceContext` involves using much the same code as in `GetVCLFrameForm` and then filling up the `TOleInPlaceFrameInfo` record.

Which now leaves us with `ShowContainerUI`. When the OLE object gets activated, the `TOleForm` deals with making sure it fits in okay. All that needs to be done here is to cater for the object being deactivated. Unfortunately the `TOleForm` doesn't spot this happening and leaves things in a bit of a state. We need to get rid of all the OLE merged menus, and also make sure that we re-accommodate any space that was made to fit in the OLE object application toolbars.



➤ *Figure 1: A TRichEdit with pictures in (for a change).*

Now that all the methods of this interface have been covered, let's see what else Listing 2 shows that we haven't looked at. Firstly, you should see that the unit this code comes from (`OleRichEdit` on this month's companion disk) implements a simple component inherited from `TRichEdit`, called `TOleRichEdit`. The only code in the component itself is in the `CreateHandle` method. After the underlying Windows rich text control window handle has been created, the component sends the control a message telling it about the dedicated OLE callback interface that can be used.

Again, because this callback object is passed across as an interface, it will be automatically destroyed when the rich edit has finished with it.

The last thing to say about the listing is that the `TRichEdit-OleCallback` destructor does an important job. When the callback object is destroyed, it notifies the `TOleForm` object in the underlying Delphi form that there is no longer an active OLE object. There is a big potential problem being avoided here. If an application developed in Delphi 3 or 4 is closed whilst you are editing an active in-place OLE object in one of these rich edits, the following sequence of events occurs. The form destroys the `TRichEdit`. The underlying Windows rich text control destroys the OLE callback object. Shortly after the `TOleForm` tries to tidy away the OLE interfaces of the rich edit, which have already gone, causing a rather nasty Access Violation.

This component can be used as a plug-in replacement for a normal `TRichEdit` and, to prove the point, there is a project on the disk called RichEdit.Dpr. This is the Delphi 3 sample project that shows the capabilities of a `TRichEdit`, but with two small changes. The `TRichEdit` has been replaced with a `TOleRichEdit` (don't forget to install this component before opening the project), and a minor amendment has been made to the main menu.

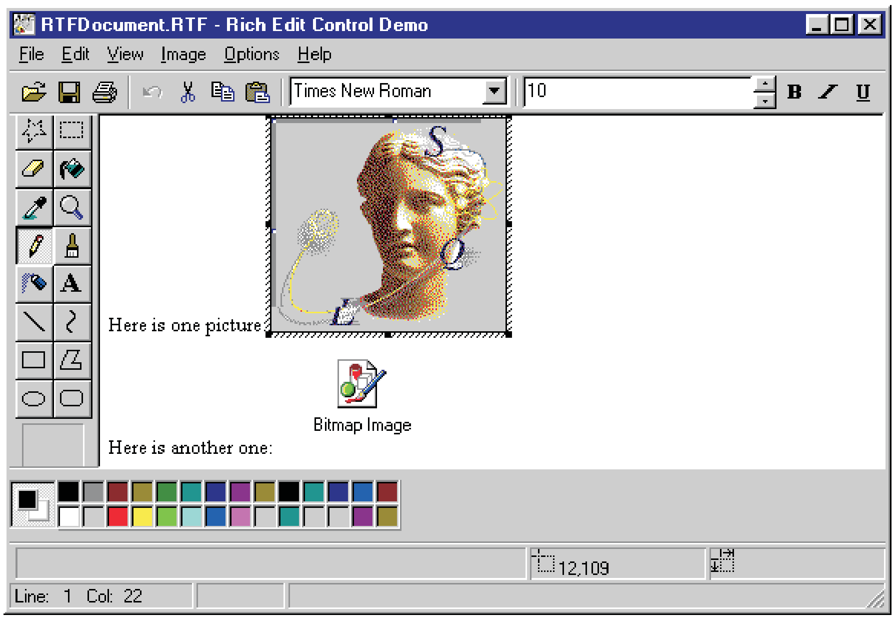
When OLE objects perform menu merging, the VCL takes note of the `GroupIndex` property of the top level menus to decide which of the original menus to leave, and which ones to hide. Menus with a `GroupIndex` of 0, 2 or 4 are left on the menu bar. I decided to leave the `File` menu with a `GroupIndex` of 0, but set all the other menus to have a `GroupIndex` of 1, to have them removed from the bar when an OLE object became active.

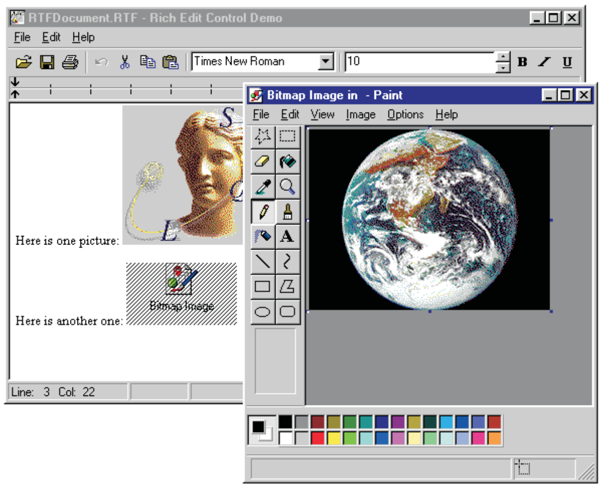
Some screenshots will verify how the rich edit deals with things. Figure 1 shows the RichEdit sample project with a simple sample RTF file open. It has one bitmap object embedded normally, and another bitmap object embedded but displayed as an icon. The file was made in WordPad. I'll leave the business of adding an Insert Object dialog into the application as an exercise for the reader. You can get most of the source from the `OleCtnrs` unit. Check the `TOleContainer` class's `InsertObjectDialog` method.

Figure 2 shows the application after double clicking the Athena image, and Figure 3 shows what happens when you double click the iconic image.

In summary, any `TRichEdit` object can do more than you may think. Prudent studying of the MSDN (and indeed other applications) can sometimes help identify what extra facilities may be available in Windows. In many cases, these extra facilities are not that hard to shoe-horn into your application. In this particular case, a new component has a total of just

➤ *Figure 2: In-place editing of an OLE object in a TRichEdit.*





➤ *Figure 3: Another OLE object in a TRichEdit being edited.*

one overridden method to get support for OLE objects in rich text controls.

Brian Long is a UK-based freelance consultant and trainer. He spends most of his time running Delphi and C++Builder training courses for his clients, and doing problem-solving work for them. The rest of his time is filled writing for this very magazine and speaking at the odd conference here and there. You can reach him at brian@blong.com