The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Strings And Streams

Q I get the impression that `TStream` objects should be used where possible, so I'd like to read and write to a text file using a `TFileStream`. However, I cannot see how I can read from the file as the `Read` procedure must know in advance how long the string is to be read. All the examples I have seen show writing strings only. It is easy reading/writing using the old procedures `ReadLn` and `WriteLn`.

A Stream objects are useful in that they provide a uniform interface to a variety of storage media, but there's no need to use them all the time. In fact, a text file is probably a good case to not use a stream. All data written to a text file is translated into a series of characters. The `Read`, `ReadLn`, `Write` and `WriteLn` routines in conjunction with a text file variable are well set up to deal with this arrangement.

Other files, where for example numerical information is stored in binary form, may be better suited to stream access. This is because the memory buffer that you write to a stream will be stored in the stream in the same binary format.

However, it is common to store text information alongside binary data in binary files, and so the question still highlights a problem. How are strings best written to streams to aid reading them back again, given that the stream read method needs to know how many bytes to read with each operation?

My answer would be to write the string length first, followed by the string data. 32-bit Delphi strings can contain potentially up to 2Gb of text, so the string length is a four-byte value. A pair of helper routines to read a string from a stream and write a string to a stream are shown in Listing 1.

```
procedure WriteStringToStream(const S: String; Stream: TStream);
var StrLen: Cardinal;
begin
  StrLen := Length(S);
  Stream.Write(StrLen, SizeOf(StrLen));
  if StrLen > 0 then
    Stream.Write(S[1], Length(S))
end;
function ReadStringFromStream(Stream: TStream): String;
var StrLen: Cardinal;
begin
  Stream.Read(StrLen, SizeOf(StrLen));
  SetLength(Result, StrLen);
  if StrLen > 0 then
    Stream.Read(Result[1], StrLen);
end;
```

➤ *Listing 1: Routines for dealing with strings in streams.*

```
procedure TForm1.Button1Click(Sender: TObject);
var S: TStream;
begin
  S := TMemoryStream.Create;
  WriteStringToStream(Edit1.Text, S);
  S.Position := 0;
  Edit2.Text := ReadStringFromStream(S);
  S.Free
```

➤ *Listing 2: Reading/writing strings from/to streams.*

You can see that the length of the string and the string data itself are dealt with separately in both cases. It is also important that the code checks if there are any characters in the string before trying to access them. If an empty string is passed in, an attempt to access the first character of the string would likely cause an Access Violation, as no memory would have been allocated for the characters.

When reading, the string length is read first and is used to set the length of the string, thereby allocating enough space for it. Then the string length is used to specify how many further bytes to read from the stream, if any.

Listing 2 shows how to write the contents of one edit control to a memory stream, then read it back into another edit control.

Optimised Working Set

Q I have noticed that when I minimise my application on Windows NT, the Task Manager shows a significant reduction in memory usage. I don't know how or why this happens, but am I able to get the same effect using code in my application, for example in a timer's event handler?

A This is indeed possible. In fact it has been discussed in *The Delphi Magazine* in Issue 39 (November 1998). In his article *Slimming The Fat Off Your Apps* Hallvard Vassbotn passed on Roy Nelson's tip of calling the `SetProcessWorkingSetSize` API in your project source file, to minimise your application's working set after all the units have initialised.

However, there is nothing stopping you calling the API regularly throughout the lifetime of your application, if you want to keep its memory consumption in check. A timer can be used for the job, with the `Interval` property set to a high value, such as 10,000 or 30,000, to trigger the `OnTimer` event every 10 or 30 seconds as you prefer. The call in the event handler would

look something like Listing 3. `GetCurrentProcess` returns a handle to the currently executing process, whilst the values of `$FFFFFFFF` request the minimum possible working set.

Note that the API only has an effect on Windows NT and 2000. In Windows 9x it is implemented but just calls `SetLastError(ERROR_CALL_NOT_IMPLEMENTED)`. Listing 3 avoids calling the two APIs if the code is not running on a suitable platform.

Hallvard mentioned in his article that you can trim the working set of the IDE, by adding a suitable unit to a package that calls `SetProcessWorkingSetSize` just after start up. If you wished you could set up a timer in the packaged unit that repeatedly trims the working set every 30 seconds. Such a unit might look like Listing 4. This unit can be added to an IDE package to keep the IDE working set trimmed, or can be added to any normal application project to keep that application's working set trimmed.

On my Windows 2000 machine, Delphi starts up having used nearly 14Mb of memory, which rapidly goes up to 18Mb with a small compilation. With the unit installed to trim the working set every 30 seconds, the memory usage is regularly chopped back down to less than 3Mb. Quite an improvement, I'm sure you will agree.

➤ *Listing 3: Keeping your working set trimmed.*

```
if Win32Platform =
  VER_PLATFORM_WIN32_NT then
  SetProcessWorkingSetSize(
  GetCurrentProcess, $FFFFFFFF,
  $FFFFFFFF)
```

➤ *Listing 4: Trimming the IDE's working set.*

```
unit TrimWorkingSet;
interface
implementation
uses
  SysUtils, Windows, ExtCtrls;
type
  TTrimmer = class(TTimer)
  public
    procedure TimerTick(
      Sender: TObject);
  end;
procedure TTrimmer.TimerTick(
  Sender: TObject);
begin
  if Win32Platform =
    VER_PLATFORM_WIN32_NT then
    SetProcessWorkingSetSize(
    GetCurrentProcess, $FFFFFFFF,
    $FFFFFFFF)
end;
var Timer: TTrimmer;
initialization
  Timer := TTrimmer.Create(nil);
  Timer.Interval := 30000;
  Timer.OnTimer := Timer.TimerTick;
finalization
  if Assigned(Timer) then
    Timer.Free;
end.
```

Copying Selected Text Between TRichEdit Controls

Q I want to copy the selected text from one rich edit control to another rich edit control, without using the clipboard. I tried:

```
dstRichEdit.Lines.Text :=
  srcRichEdit.SelText;
```

but this copies the selected text without formatting. How can I get the formatting as well without destroying what is already on the clipboard?

A In my article *TRichEdits And Embedded Objects* in Issue 52, I discussed how rich edit controls can make use of the `IRichEditOleCallback` interface in order to support embedded OLE objects. This interface is not defined in the units supplied with Delphi (at least up to version 5), so I had to define it myself. The original C++ definition of the interface comes from a header file RichOLE.h, which has no equivalent in Delphi's import units. Once you define the interface, you can implement it in an object and pass a reference to it to the rich edit control with the `EM_SETOLECALLBACK` message.

Another interface defined in that header that is consequently not defined in Delphi is `IRichEditOle`. This interface can be used to help answer this question and so, again, we will need a Delphi definition of it. This month's disk contains a unit called RichOle.pas that contains the `IRichEditOleCallback` and `IRichEditOle` interfaces, along with some constants and types used by the parameters of the interface methods (see Listing 5).

Having now got a definition of `IRichEditOle`, what does it allow us to do? Well to start with, the rich edit control already implements this interface, so we need to ask the control for a reference to it with the `EM_GETOLEINTERFACE` message. Once we have access to the interface, two key methods will be used to help answer our question: `GetClipboardData` and `ImportDataObject`.

When applications support both the Windows clipboard and inter-application OLE-based drag and drop, they represent shareable data by way of a data object (represented by the `IDataObject` interface). The data object can manage data in a variety of formats stored in a variety of storage media.

For example, when Microsoft Word copies some text into the clipboard, it first gives the data (in a number of formats) to a data object. This data object is then effectively placed in the clipboard. When you drag some text from Microsoft Word to another application, it again gives the data to a data object to maintain.

The `IRichEditOle` interface allows us to get a data object representing any range of text in the rich edit control using its `GetClipboardData` method. This returns a reference to an `IDataObject` interface that represents the requested data. Note that this data object is not given to the clipboard (though it could be) but is merely returned from the method call.

Having got the data object representing some text (and formatting) in the source rich edit, we now need to get it into the destination rich edit. This can be done by getting access to the destination rich edit's `IRichEditOle` interface and passing the data object to its `ImportDataObject` method. The destination rich edit will absorb all the data from the data object and the job will be done.

OK, enough preamble. Let's get on with it. RichEditCopying.dpr is a sample project on this month's disk. It has a source and destination rich edit (`reSrc` and `reDest` respectively), along with a button that copies the source rich edit selection to the destination rich edit. The button's `OnClick` event handler is shown in Listing 6.

```
type
  // Structure passed to GetObject and InsertObject
  TREObject = record
    cbStruct: DWord; // size of structure in bytes
    cp: Longint;     // character position of object
    ClsID: TClsID;   // class identifier of object
    pOleObj: IOleObject; // OLE object interface
    pStg: IStorage;  // associated storage interface
    // associated client site interface
    pOleSite: IOleClientSite;
    sizel: TSize;    // size of object (may be 0,0)
    dvaspect,        // display aspect to use
    dwFlags,         // object status flags
    dwUser: DWord;   // user-defined value
  end;
  IRichEditOle = interface
    ['{00020D00-0000-0000-C000-000000000046}']
    procedure GetClientSite(
      out lplpOleSite: IOleClientSite); stdcall;
    function GetObjectCount: Longint; stdcall;
    function GetLinkCount: Longint; stdcall;
    function GetObject(iObj: Longint; out reobject:
      TREObject; dwFlags: DWord): HResult; stdcall;
    function InsertObject(const reobject: TREObject):
      HResult; stdcall;
    function ConvertObject(iObj: Longint; const clsidNew:
      TClsId; lpStrUserTypeNew: lpCStr): HResult; stdcall;
    function ActivateAs(const clsId, clsIdAs: TClsId):
      HResult; stdcall;
    function SetHostNames(lpstrContainerApp,
      lpstrContainerObj: lpCStr): HResult; stdcall;
    function SetLinkAvailable(iObj: Longint; fAvailable:
      Bool): HResult; stdcall;
    function SetDvaspect(iObj: Longint; dvaspect: DWord):
      HResult; stdcall;
    function HandsOffStorage(iObj: Longint): HResult;
      stdcall;
    function SaveCompleted(iObj: Longint; stg: IStorage):
      HResult; stdcall;
    function InPlaceDeactivate: HResult; stdcall;
    function ContextSensitiveHelp(fEnterMode: Bool):
      HResult; stdcall;
    function GetClipboardData(const chrg: TCharRange; reco:
      DWord; out dataobj: IDataObject): HResult; stdcall;
    function ImportDataObject(dataobj: IDataObject; cf:
      TClipFormat; hMetaPict: HGlobal): HResult; stdcall;
  end;
```

➤ *Listing 5: The IRichEditOle interface.*

The first step is to get the source rich edit's OLE interface. If this is successful, the selected text in the source rich edit is recorded in a `TCharRange` record variable, which needs starting and ending characters. The rich edit's `SelStart` and `SelLength` properties are used to fill in the record fields.

The next step is to ask for a data object that represents information being copied from the source rich edit control (rather than being cut, for example). If the data object is returned it needs to be passed to the destination rich edit. To do this, we need the destination rich edit's OLE Interface. If it is returned without problem, the data object is passed in to the `ImportDataObject` method. Figure 1 shows the program running, where a selection has just been copied across.

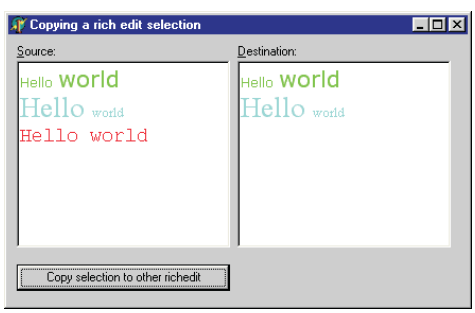
You can see a more in-depth use of data objects in my *Dragging And Dropping* article in this issue.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  reoSrc, reoDest: IRichEditOle;
  DataObj: IDataObject;
  CharRange: TCharRange;
begin
  //Copies text but not formatting
  //  reoDest.Lines.Text := reoSrc.SelText;
  //Copies text and formatting, but uses the clipboard
  //  reoSrc.CopyToClipboard;
  //  reoDest.PasteFromClipboard;
  //Copies any range, with formatting, and
  //without destroying the clipboard contents
  reoSrc.Perform(EM_GETOLEINTERFACE, 0, LParam(@reoSrc));
  if Assigned(reoSrc) then begin
    CharRange.cpMin := reoSrc.SelStart;
    CharRange.cpMax := reoSrc.SelStart + reoSrc.SelLength;
    reoSrc.GetClipboardData(CharRange, RECO_COPY, DataObj);
    if Assigned(DataObj) then begin
      reoDest.Perform(EM_GETOLEINTERFACE, 0, LParam(@reoDest));
      if Assigned(reoDest) then
        reoDest.ImportDataObject(DataObj, 0, 0);
    end
  end
end;
```

➤ *Listing 6: Using a data object to copy information between rich edits.*

Rich Text To Microsoft Word

Q I have an app where I am saving some rich text data entered via a `TDBRichEdit` field. I want to transfer it to Microsoft Word, but am having difficulty getting it to work correctly. I've tried passing the field's `AsText` property to the appropriate Word method, but I get the raw RTF showing, with all the curly brackets and tags. Next, I tried copying the field to the clipboard, by assigning the field's `AsText` property to the `ClipBoard` object's `AsText` property. The goal was to paste the RTF data into the Word document by using Word's `PasteSpecial` method, specifying RTF format. This idea was thwarted because it reported that the format was not available. Am I missing something obvious?



➤ *Figure 1: Copying a rich edit selection.*

A I am no expert with Word, so I don't know if there is a simple way of achieving this directly, but I can show you how to do it using the clipboard. Clearly, if the application copies data to the clipboard, then pastes that data into Word, any data placed in the clipboard by the user will be lost. Without using a temporary file (which I feel would be slower), I can't think of a way of avoiding the clipboard. Assuming this disadvantage is not a problem, let's look at what needs to be done.

Firstly, you need to get a data object representing the RTF data from the rich edit that you want copied. This can be done in exactly the same way as in the previous *Delphi Clinic* entry. The only difference is that all the text from the source rich edit needs to be copied this time. To indicate this the `TCharRange` record is given a minimum character position of 0 and a maximum of -1.

To allow Word to paste RTF from the clipboard, the data object must be given to the clipboard.

This is done by passing it to the `OleSetClipboard` function. Assuming the clipboard is not being held open by an application, this function should return a successful `HResult` value.

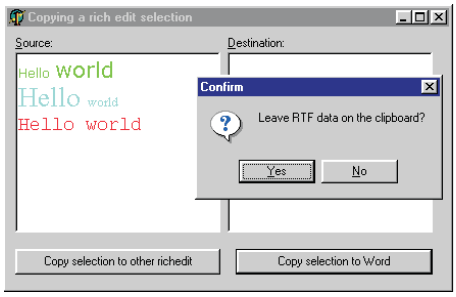
Now that the data object is in the clipboard, the content of the rich edit control can be accessed from the clipboard in various ways, including unformatted text and formatted text (RTF).

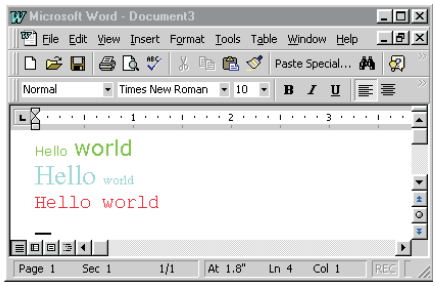
Automation can then be used to connect to Word and get the clipboard data pasted into a document. You can see all this happening in Listing 7, from the RichEditCopying2.dpr project (the same as RichEditCopying.dpr but with an extra button to copy the entire content of the source rich edit to Word via the clipboard).

After the Automation code, a `try..finally` statement ensures that some form of tidying up is done. Assuming the data object is still on the clipboard (checked by `OleIsCurrentClipboard`), the user is asked if they want the data represented by the data object left in the clipboard. This is much the same as when you close Word. If it feels there is some non-trivial data on the clipboard it asks if you want to keep it there or clear it (see Figure 2).

➤ *Listing 7: Copying RTF text into Word via the clipboard.*

```
procedure TForm1.Button2Click(Sender: TObject);
var
  reoSrc: IRichEditOle;
  DataObj: IDataObject;
  CharRange: TCharRange;
  MSWord: Variant;
const
  wdPasteRTF = 1;
begin
  reSrc.Perform(EM_GETOLEINTERFACE, 0, LParam(@reoSrc));
  if Assigned(reoSrc) then begin
    CharRange.cpMin := 0;    //Select all text
    CharRange.cpMax := -1;
    //Place data object for rich edit content on clipboard
    reoSrc.GetClipboardData(CharRange, RECO_COPY, DataObj);
    OleCheck(OleSetClipboard(DataObj));
    try
      try
        MSWord := GetActiveOleObject('Word.Application');
      except
        MSWord := CreateOleObject('Word.Application');
      end;
      MSWord.Visible := True;
      MSWord.Documents.Add;
      MSWord.Selection.PasteSpecial(DataType := wdPasteRTF)
    finally
      if OleIsCurrentClipboard(DataObj) = S_OK then
        if MessageDlg('Leave RTF data on the clipboard?',
          mtConfirmation, [mbYes, mbNo], 0) = mrYes then
          OleCheck(OleFlushClipboard)
        else
          OleCheck(OleSetClipboard(nil))
    end
  end
end;
```





➤ *Figure 2: Copying rich text to Microsoft Word.*

To clear the clipboard, the code passes `nil` to `OleSetClipboard`. To leave the data on the clipboard, but remove the data object maintaining the data, you call `OleFlushClipboard`. This allows the program to be terminated, but still leaves the data available in all its formats for more paste operations.

API problem

Q I am worried about the `WaitForSingleObject` API. I use it to detect when a launched process terminates by waiting for the process handle to become signalled. A process handle becomes signalled when the corresponding process is terminated. The problem occurs with just a few applications including Microsoft Paint (PBrush.exe) and Windows Explorer. I launch them, they start running, and `WaitForSingleObject` immediately reports that the process handle has become signalled. I don't know how to find out what is wrong. Can you help?

A This one certainly looks confusing at first. A sample project called AppLauncher.dpr on the disk reproduces this problem. It uses a utility routine based on those shown in the *CreateProcess Alert* entry from *The Delphi Clinic* in Issue 51. The `RunCommand` routine is shown in Listing 8.

As well as the command-line string and parameters, the routine takes references to two parameterless procedure methods. The first is called immediately after launching the application, whilst the second is called when the launched application terminates.

To make it easy to see what happens, I decided to minimise the application launcher after starting the secondary program. When the launched program terminates, the application restores itself. The reason for minimising the application is that the calling thread is frozen by a call to `WaitForSingleObject` until either the relevant object becomes signalled or the wait operation times out. This leads to a hung user interface. Minimising first means this UI problem is effectively masked from the user.

Methods already exist in the `Application` object to minimise and restore the application, and it so happens that both methods are parameterless procedure methods. This means I can pass references to these methods when I call `RunCommand`, thereby saving me the trouble of setting up new methods and passing references to those (see Listing 9).

As mentioned, if you use this application to launch either PBrush.exe or Explorer.exe, the application will minimise and then

```
type
  TWaitProc = procedure of object;
  TBeforeWaitProc = TWaitProc;
  TAfterWaitProc = TWaitProc;
procedure RunCommand(const Cmd, Params: String;
  BeforeWait: TBeforeWaitProc; AfterWait: TAfterWaitProc);
var
  SI: TStartupInfo;
  PI: TProcessInformation;
  CmdLine: String;
begin
  FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);   //Set mandatory record field
  //Ensure Windows mouse cursor reflects launch progress
  SI.dwFlags := StartF_ForceOnFeedback;
  CmdLine := Cmd;       //Set up command line
  if Length(Params) > 0 then CmdLine:=CmdLine+#32+Params;
  //Try and launch child process. Raise exception on failure
  Win32Check(CreateProcess(nil,PChar(CmdLine),nil,nil,False,0,nil,nil,SI,PI));
  try
    //Wait until process has started its main message loop
    WaitForInputIdle(PI.hProcess, Infinite);
    if Assigned(BeforeWait) then BeforeWait;
    WaitForSingleObject(PI.hProcess, Infinite);
    if Assigned(AfterWait) then AfterWait;
  finally
    CloseHandle(PI.hThread);
    CloseHandle(PI.hProcess);
  end
end;
```

➤ *Listing 8: A command-line executor.*

```
procedure TForm1.btnLaunchClick(
  Sender: TObject);
begin
  RunCommand(edtCommand.Text,
    edtParams.Text,
    Application.Minimize,
    Application.Restore)
end;
```

➤ *Listing 9: Launching programs.*

immediately restore itself, despite the fact that the launched program (Paint or Explorer) is still running. Figure 3 shows the application launcher just after launching Paint: you can see it has restored itself.

So what is the problem? I tried looking for problem reports on `WaitForSingleObject` on the MSDN CD, but found `WaitForSingleObject` has no problem waiting for process handles to become signalled.

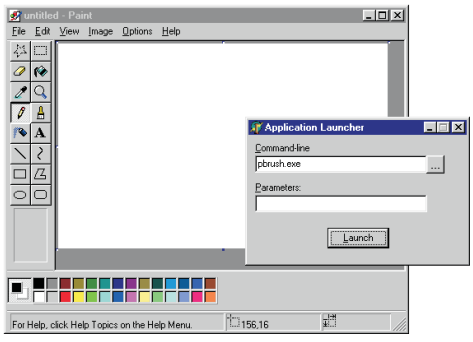
Next, I tried using the file access monitor from www.sysinternals.com. I started it before launching the application and observed the information that it produced. This showed the problem immediately.

Let's take Microsoft Paint first. To run the application, you run PBrush.exe. However, PBrush.exe is not Microsoft Paint, but is a trivial application in the Windows directory whose sole task is to launch a copy of Paint, which is in fact in MSPaint.exe, which lives in C:\Program Files\Accessories, with WordPad. So, when you launch PBrush.exe, it will start up, launch a copy of MSPaint.exe, and then terminate, thereby making its process handle become signalled.

The situation is similar with Explorer. When you run a new copy of Explorer it checks to see if it is already running. Since it definitely will be (Windows Explorer acts as the Windows shell as well as a file browser), the new copy of Explorer communicates with the already running version, tells it what it needs to do, then terminates. The Explorer window that gets displayed comes from the original copy of Explorer (the Windows shell). So the freshly launched second copy of Explorer terminates immediately, and its process handle becomes signalled.

You will need to test these things whilst developing your application to ensure that the applications you launch actually stay running. If they don't, more investigation may be necessary, and possibly alternative strategies devised. For Paint, the solution is to launch MSPaint.exe not PBrush.exe.

For Explorer, the job is more difficult. You'll need to identify the window that gets launched using `FindWindow` and `EnumWindows`. Before launching Explorer you should get a list of all existing windows, using `EnumWindows`. After Explorer terminates you should make another list and concentrate on the new additions. `FindWindow` can verify whether the window still exists.



➤ *Figure 3: The application launcher thinks Paint has terminated.*

ActiveForm Destruction

Q I've been creating ActiveForms in Delphi and have a question about when these are being freed. I added a simple `ShowMessage` call to the ActiveForm's `OnDestroy` event to show when it was being triggered. However, the message never appears when the control is used, either on a web page or from within another Delphi app. I had assumed that when a control was used on a web page, the browser would destroy it when no longer required. I also assumed the control would be destroyed when a Delphi form hosting the control was freed. Is this ActiveX control not being destroyed or should I do something differently?

A My initial response to this question, whilst valid in its content, turned out to be unrelated to the problem upon investigation. I thought the call to `ShowMessage` was the root of the problem. As I described in the *Active OLE Object* entry in *The Delphi Clinic* from Issue 46, if the application is in the process of being terminated (and therefore `Application.Terminated` is `True`) modal forms will not display. The implementation of `ShowModal` checks the property and skips out if it is to be `True`. Calls like `ShowMessage` and `MessageDlg` are based on modal forms, so will not display if the application is terminated.

Whilst this is true, it has nothing to do with this problem. Calls to `Application.MessageBox` in

an `OnDestroy` event handler are also ignored, despite it not using Delphi modal forms. The problem actually lies rather deeper.

When you're creating an ActiveForm, it presents itself in the Form Designer just like a normal form. There are a number of events available on the Object Inspector's `Events` page but some of them need to be avoided (see below).

Some of the events are represented by entries in an events interface that you'll find in the Type Library editor. Any host program can implement that events interface, thereby making itself an event sink, and should be notified when the relevant events fire in the ActiveForm control.

Table 1 shows how an ActiveForm deals with each of the events it publishes. As you can see, the ActiveForm code in the unit generated by the wizard sets several event handlers (such as `OnClick` and `OnDblClick`) to specific methods that exist in the class. This is done in the implementation of the `Initialize` method. Those methods check to see if the host application has set up an event sink for the events interface (which includes each of the events that are explicitly set up). Assuming it has, the host event is triggered.

This works fine for most of the events except `OnCreate` and `OnDestroy`. The `OnCreate` event will have already fired in the ActiveForm before `Initialize` executes and sets up the method that triggers the event in the host. It is therefore pointless setting up an `OnCreate` event handler in the host application. You should set it up in the ActiveForm itself.

Also pointless is surfacing the `OnDestroy` event handler to the host. By the time the `OnDestroy` event is triggered (as the ActiveForm is being destroyed), the host application will have disconnected its event sink from the ActiveForm. Consequently, the method in the ActiveForm that checks to see if there is an event sink for its events interface will not find one. An `OnDestroy` event handler in the host application will therefore not fire. Also, the `OnDestroy` event in the ActiveForm gets overwritten by the `Initialize` method, so it is not sensible to set an event handler there either.

From this, it seems you cannot have an operative `OnDestroy` event handler for an ActiveForm. Also, an `OnCreate` event handler must be set up in the ActiveForm, not in the host application. This implies that neither `OnCreate` nor `OnDestroy` should be in the events interface of an ActiveForm. I've reported this incorrect behaviour to Inprise.

The final point to make is that despite `OnDragOver` and `OnDragDrop` not being part of the events interface, these events still fire in the client (rather than the ActiveForm). However, this is really due to the VCL class used to represent an imported ActiveX. `TOleControl` defines `OnDragOver` and `OnDragDrop`. So, when your ActiveForm is imported into another development tool you will probably not get these events.

➤ *Table 1: ActiveForm events and where they fire.*

ActiveForm Event	Behaviour Of An ActiveForm	Where Event Handler Executes
OnActivate	Overwrites event to trigger it in the host	Host
OnClick	Overwrites event to trigger it in the host	Host
OnContextPopup	Leaves event untouched	ActiveForm
OnCreate	Overwrites event to trigger it in the host	ActiveForm
OnDblClick	Overwrites event to trigger it in the host	Host
OnDeactivate	Overwrites event to trigger it in the host	Host
OnDestroy	Overwrites event to trigger it in the host	Nowhere
OnDragDrop	Leaves event untouched	Host
OnDragOver	Leaves event untouched	Host
OnKeyDown	Leaves event untouched	ActiveForm
OnKeyPress	Overwrites event to trigger it in the host	Host
OnKeyUp	Leaves event untouched	ActiveForm
OnMouseDown	Leaves event untouched	ActiveForm
OnMouseMove	Leaves event untouched	ActiveForm
OnMouseUp	Leaves event untouched	ActiveForm