The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Where Is *My Documents*?

Q I understand that Microsoft advises developers to have applications store documents in the *My Documents* folder (or at least default to that directory). I gather this is particularly important with the advent of Windows 2000. How can I find out where the current user's *My Documents* folder is located?

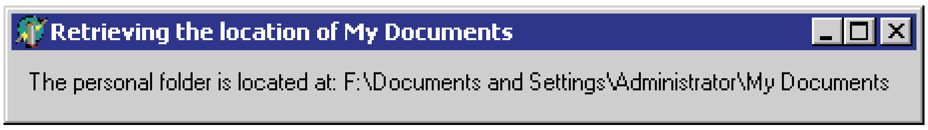
A This is a job for the Shell API. The user's personal documents folder (*My Documents*) is one of a number of special folders understood by the shell. We have seen uses of the shell `SHGetSpecialFolderLocation` API to get the location of some of these special directories in *The Delphi Clinic* in Issues 39 (*Creating Folders and Shortcuts*), 47 (*Temporary Internet Files*) and 52 (*Computer Picker*).

`SHGetSpecialFolderLocation` returns a pointer to an item identifier list (a PIDL) which will ultimately need to be freed by being passed to the `Free` method of an `IMalloc` interface reference. Before freeing it, you can turn the PIDL into a path with a call to `SHGetPathFromIDList`. Listing 1 shows a routine that does the job. A sample project called MyDocs.dpr is on the disk this month and Figure 1 shows it using the code in Windows 2000.

Incidentally, the code shown in Issue 52 (based on code from a MIDAS property editor) does not free the PIDL. Listing 1 shows how the `IMalloc` reference should be correctly obtained and used to free a PIDL. I have reported the property editor resource leak as a bug.

```
uses
  ShellAPI, ShlObj, ActiveX;
function GetMyDocuments: String;
var
  PIDL: PItemIDList;
  MyDocsC: array[0..MAX_PATH] of Char;
  Malloc: IMalloc;
begin
  SHGetMalloc(Malloc);
  if SHGetSpecialFolderLocation(Application.Handle,
    CSIDL_PERSONAL, PIDL) = NOERROR then
  try
    if SHGetPathFromIDList(PIDL, MyDocsC) then
      Result := MyDocsC
    else
      raise EInvalidOp.Create('Cannot find personal documents folder')
  finally
    Malloc.Free(PIDL)
  end
end;
```

➤ *Listing 1: Finding the location of the My Documents folder.*



Retrieving the location of My Documents

The personal folder is located at: F:\Documents and Settings\Administrator\My Documents

➤ *Figure 1: Finding the My Documents folder in Win 2000.*

Getting UNC File Names

Q How do I translate a drive-based filename to a UNC name?

A UNC is the **U**niversal **N**aming **C**onvention, which describes drives in terms of their server and share names, as in `\\<servername>\<sharename>`. The Delphi Run-Time Library has had a routine that will do the job since Delphi 2. Pass your filename into the `ExpandUNCFileName` function and, if the file is located on a drive that is a network resource (as opposed to a local drive), the drive portion of the filename will be converted to UNC.

If you also wanted local drive references translated to UNC, you would need to do that part yourself, using knowledge of the computer name (which can be extracted with the `GetComputerName` Win32 API, as discussed in *The Delphi Clinic* in Issue 49) and the active share names.

Adding Tool Buttons

Q I am trying to write a toolbar customisation facility for our application. My problem arises when I try to add a new button to the `TToolBar`. The help says there is a method for `TToolButton` called `SetToolBar`, but this is not a public function. I'm not really *au fait* with messing around with the VCL and was wondering if there is a way to gain access to this function?

A This is a case where you are looking for a public method, but the one that does the job is protected. Consequently, you are going to have to take advantage of the details of the definition of the protected part of a class in Object Pascal to get the job done. Items in the protected section of a class can be accessed by classes inherited from themselves, and also by any code in the same unit as the class.

So, to open the door to the protected section of `TToolButton`, define a class inherited from it (without defining any new fields or methods) in the unit where you are creating tool buttons. Next, perform a static typecast to treat your tool button object as if it were the new class (which is identical to TToolButton, due to it having nothing added to it). Now that you are accessing a tool button under the guise of the new class in the current unit, you will be able to access any protected members that you like. Listing 2 shows the idea.

Modifying The VCL Source

Q I have come across the TMaskEdit bug (see *Edit Mask Hang Bug* in *The Delphi Clinic* from Issue 18) whereby I get weird behaviour with some BIOS versions. The symptoms are flashing Num Lock and Scroll Lock lights, and/or very slow (or complete lack of) keyboard response. The solution is a simple change to the protected SetCursor method within TCustomMaskEdit in the Mask unit.

I am having difficulties recompiling Mask.pas in isolation under Delphi 5. I have followed suggestions of placing the unit in a separate directory and pointing my compiler search path to it before Delphi's Lib directory. Attempting to recompile causes the compiler to fail with the message: *Grids.pas was compiled with an older version of Mask.pas*. It would seem from this I need to recompile the entire VCL, which I am loath to do. How can I compile a fixed version of the Mask unit that I can then use to replace the old version in my Lib directory?

A The problem may be occurring because your project has other units used which are causing interdependency problems for the compiler with only that source file available. When trying to solve the problem I copied the unit to a new folder, saved a fresh, new, project in there too and added the Mask.pas source unit to the project. I built the project and successfully obtained a Mask.dcu file. I then applied the changes suggested back in Issue 18, recompiled the project and got a differently sized DCU.

However, if this DCU is to be used as a replacement for the one in Delphi's Lib directory, it would be wise to ensure it is compiled with exactly the same options as the original. To accomplish this, I deleted the project's CFG file. This file is generated by Delphi 4 and 5 and acts as a command-line compiler configuration file, containing command-line switches to give exactly the same set of compiler/linker switches at the command line as in the IDE. With the CFG file gone, a command-line of:

```
DCC32 -B -$D- -$L- Project1.dpr
```

will rebuild the project (and so recompile the Mask unit) using standard options: no debugging or line number information, all other switches as their defaults. The resultant DCU file can be copied over the one in the Lib directory (but don't forget to backup the original version first, just in case).

COM Terminology Problem

Q I use TClientDataSet components in my application because they provide a convenient memory-resident table for lookup lists. Using a TClientDataSet requires DBCLIENT.DLL to be distributed with the application. Recently, my client application refused to work because it could not find DBCLIENT.DLL. After some tracking down I found that this DLL has a registry entry and it was incorrect.

I concluded from this that DBCLIENT.DLL is in fact an OCX. What are the advantages of an OCX over a DLL and how do you create an OCX from a DLL? Also, what is an OCX's relationship to ActiveX?

A The use of TClientDataSet as a convenient BDE-less database engine indeed requires DBCLIENT.DLL to accompany the application when built with Delphi 3 or 4, and MIDAS.DLL to accompany the application when built with Delphi 5 or later.

DBCLIENT.DLL and MIDAS.DLL are DLL files, but they act as in-process COM servers. An in-process COM server is a DLL that will be loaded into the process address space of the application that makes use of it, and which contains implementations of one or more COM objects. In order for COM objects to be accessible, details about them must be stored in the Windows registry.

To get details about the COM objects (and all the new interfaces) in an in-proc COM server into the registry, you either pass it as a command-line parameter to the Inprise TRegSvr.exe console application or the Microsoft RegSvr32.exe console application. You can also do it programmatically, using code based on the *OCX Deployment* entry in *The Delphi Clinic*, Issue 19.

A DLL exports subroutines, whereas a COM server makes objects available in a language-independent manner, so the key benefit of a COM server is that it gives applications access to objects, rather than just functions and procedures. To make an in-process COM server in Delphi, choose File | New... and from the ActiveX page choose an ActiveX Library. If you look closely at what gets generated, you will see that you have a DLL project in front of you that exports four routines. Those four routines are all implemented by the ComServ unit which is in the uses clause. By exporting these routines, the DLL has conformed to the rules of being an in-proc COM server.

COM objects make as much of their implemented functionality available as they choose by defining it in terms of interfaces. An interface describes a set of routines that will be implemented by one (or more) COM objects.

➤ *Listing 2: Defining an access class to access the protected section of a TToolButton.*

```
type
  TToolButtonAccess = class(TToolButton);
...
TToolButtonAccess(MyToolButton).SetToolBar(MyToolBar)
```

Simple COM objects may choose to implement just one interface. More complex COM objects will choose to implement more than one.

An Automation server is a COM object that implements a specific interface (`IDispatch`) in order for it to work in a prescribed manner. ActiveX controls are nothing more than COM objects that implement a different predefined list of interfaces required for them to work as expected (as visual controls that can be used in many development tools). In fact, ActiveX controls have around 18 interfaces to implement, including `IDispatch`, which makes ActiveX controls special cases of Automation servers.

ActiveX controls were historically called OLE Custom Controls in a previous incarnation. The term 'OLE Custom Control' was abbreviated (in some bizarre fashion) to OCX, so what we now refer to as an ActiveX control used to be referred to as an OCX. It is quite common for the DLL that hosts an OCX or ActiveX (or indeed any COM object for that matter) to be given an OCX file extension, but this is not necessary. Clearly DBCLIENT.DLL has not been given such an extension.

In summary, DBCLIENT.DLL is not an OCX, but could be given an OCX file extension. In fact, it is an in-proc COM server implementing one or more COM Objects. The benefit of an in-proc COM server over a DLL is that objects can be made available as opposed to just routines. A DLL can become an in-proc COM server by exporting the four key routines from the `ComServ` unit, and then having COM objects defined within it. Once the COM server is registered, it is ready for use.

Adding ActiveX/ ActiveForm Properties

Q I have been making a number of ActiveX controls and ActiveForms. I would like to define new properties for them, which show up in the Object Inspector (or equivalent) in whatever development tool they are installed into. How do I accomplish this?

A The implementation of an ActiveX control or ActiveForm already has some nice generic code that saves and loads all the existing properties. What you need to do is hook into that system to achieve your goal.

In a VCL-generated ActiveX or ActiveForm, there are two classes involved. One is the ActiveX class, which is the COM object that implements all the interfaces required by an ActiveX host. This class inherits from `TActiveXControl` (a derivative of `TAutoObject`, the class used for Automation objects) and will ultimately interact with applications that host the ActiveX control. The other class involved is the visual control that is represented by the ActiveX control, which must be something inherited from `TWinControl`.

So, to turn a `TWinControl` component into an ActiveX control, the ActiveX project contains a COM class inherited from `TActiveXControl` that implements all the required interfaces to do the job.

The class factory used for ActiveX controls is `TActiveXControlFactory`. ActiveForm controls use a simple derivative called `TActiveFormFactory`. An instance of the appropriate class factory is created in the initialisation section of the generated unit that contains the ActiveX or ActiveForm. It has two parameters defined to take the ActiveX class (`ActiveXControlClass`) and the represented component class (`WinControlClass`).

When you make an ActiveX control in Delphi, it asks which `TWinControl` component class you would like to turn into an ActiveX control. This becomes the `WinControlClass` parameter for the class factory. `ActiveXControlClass` is set to the class that is defined in the generated unit, whose name defaults in the ActiveX control wizard to the `TWinControl` class with an `X` suffix.

In the case of an ActiveForm class, things are the other way round. The `ActiveXControlClass` parameter is set to the VCL `TActiveFormControl` class (from the `AXCtrls` unit). The `WinControlClass` parameter is set to the form class, as defined in the generated unit.

When the class factory is asked for an ActiveX control object, it creates an instance of the class represented by `ActiveXControlClass` and returns it. When the `ActiveXControlClass` object gets created, it creates an instance of the `WinControlClass` object as a child of the ActiveX container (the window that hosts the ActiveX control).

As you develop an ActiveX control or an ActiveForm, you can define properties in the custom interface that is visible in the type library editor. In the case of an ActiveX control, this interface is implemented in the `ActiveXControlClass` COM class that is defined in the generated unit. This makes sense, as that is the object that will be liaising with the ActiveX container.

In the case of an ActiveForm, things are different, because of the converse situation laid out above. Because the class being defined in the generated unit is the `WinControlClass`, that is the class that implements the custom `IDispatch`-based interface. This is odd as this is not the real COM Object.

However, the base COM object in an ActiveForm scenario, `TActiveFormControl` manages to overcome this possible interface location issue by having a special implementation of the `IUnknown` method `QueryInterface`. Whenever an interface is queried for, the COM object first checks whether it is supported by the actual form object (the one that implements the `IDispatch`-based interface). This way, the container can find all the interfaces it requires.

When an ActiveX/ActiveForm is loaded or saved, the container of the ActiveX control does this by using the `IPersistPropertyBag`, `IPersistStreamInit` or `IPersistStorage` interfaces implemented by the `ActiveXControlClass`. In the case of `IPersistPropertyBag`, the ActiveX control iterates through each read/write property in the `IDispatch`-based interface that describes the ActiveX control, reading/writing the IDispatch property from/to an IPropertyBag interface implemented by the container.

In the cases of IPersistStreamInit and IPersistStorage, the ActiveX control is given either an IStream interface or an IStorage interface. The ActiveX code takes the given IStream, or in the case of an IStorage obtains an IStream from it. It passes it to a TOleStream wrapper object and then uses standard VCL component reading/writing methods to load/store the underlying VCL TWinControl descendant object in the stream. Those methods work on the basis of published properties.

This means that to have a new property available for your ActiveX control and for it to be saved and loaded where necessary, you must do two things. You must publish a property in the WinControlClass that is being represented as an ActiveX control, and then you must add a new property to the IDispatch-based interface describing your ActiveX control (normally done using the type library editor). The interface property reader and writer routines must be coded to access the property in the WinControlClass. Strictly speaking, the VCL published property and the COM property do not need to have the same name, but it keeps things consistent if they do.

Accomplishing these two goals is simplest with an ActiveForm. The WinControlClass is the actual form class, so the property must be published in that class. The interface that you edit in the type library editor is implemented by this class, so the property reader and writer will also get defined here. The implementations of these methods will simply access the corresponding published property.

A sample ActiveX library called ActiveX.Dpr is on the disk. This project contains an ActiveForm (called ActiveFormX) which requires a new textual property. The property will be called MyNewFormProp, and will be used to surface the caption of a label that is dropped on the form. The first thing to do is add in the implementation of a published form property (see Listing 3).

```
TActiveFormX = class(TActiveForm, IActiveFormX)
  Label1: TLabel;
private
  ...
  { My new methods }
  // Published property reader
  function GetMyNewFormProp: String;
  // Published property writer
  procedure SetMyNewFormProp(const Value: String);
  ...
published
  property MyNewFormProp: String
    read GetMyNewFormProp write SetMyNewFormProp;
end;
...
function TActiveFormX.GetMyNewFormProp: String;
begin
  Result := Label1.Caption
end;
procedure TActiveFormX.SetMyNewFormProp(const Value: String);
begin
  Label1.Caption := Value
end;
```

➤ *Listing 3: Adding a published VCL property to an ActiveForm.*

```
TActiveFormX = class(TActiveForm, IActiveFormX)
...
protected
  ...
  function Get_MyNewFormProp: WideString; safecall;
  procedure Set_MyNewFormProp(const Value: WideString); safecall;
  ...
end;
...
function TActiveFormX.Get_MyNewFormProp: WideString;
begin
  Result := MyNewFormProp
end;
procedure TActiveFormX.Set_MyNewFormProp(const Value: WideString);
begin
  MyNewFormProp := Value
end;
```

➤ *Listing 4: Adding a COM property to an ActiveForm.*

The next step is to add a new COM property to the interface. Choose View | Type Library, select the IActiveFormX interface and ask for a new property (either with the tool buttons or by right clicking). Give it a name of MyNewFormProp and a type of either WideString or BSTR, depending which is available. This will be dictated by whether the Language setting on the Type Library page of the Tools | Environment Options... dialog is set to Pascal or IDL.

When you press the Refresh button, a COM property reader and writer routine will be manufactured. These need to be implemented to access the VCL property, as shown in Listing 4. Notice that the VCL property reader and writer routines are named subtly differently to the COM reader and writer routines. The COM versions always have an underscore character in their names.

This fairly straightforward process gets you a new ActiveForm property. With an ActiveX control, it's a bit trickier, since the WinControlClass that needs the new VCL property is not the one being defined in the generated source unit.

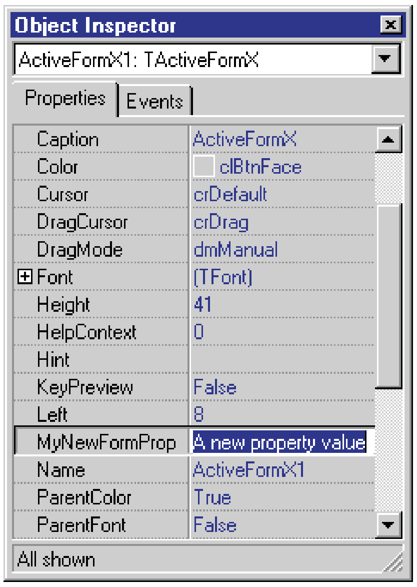
To show how to accomplish the same task with an ActiveX control, the ActiveX library project on the disk also has an ActiveX control defined in it. This is based on a standard TButton, but could be based on your own custom component class. If it is a custom component, it will be easy to add the published VCL property directly into that class. If you are using a stock VCL component, then you'll need to write a new inherited class: you will need to make changes to the code generated by the ActiveX control wizard.

In the generated ActiveX class, the VCL component type is referenced three times (see Listing 5). If you are using a stock VCL class (as Listing 5 is), you'll need to change these references for the new inherited class that we will write.

The inherited class (which will be called `TMyButton`) needs to be declared above the ActiveX class (`TButtonX` in this case). The new property will be similar to the one we added to the ActiveForm (but called `MyNewButtonProp`) and so the new class will look like Listing 6.

All that is now left is to change the three references of `TButton` to `TMyButton`, and add a new COM property to the interface (`IButtonX` in this case), just as before. This time, however, the COM property reader and writer routine will access the property in the button control (see Listing 7).

Another project accompanies the ActiveX library project on the disk. This one assumes that you have imported the ActiveX controls onto Delphi's component palette as it has an instance of the `ButtonX` and `ActiveFormX` object on the form. The custom property of each of these two controls has had a value assigned to it. You will find that the value is stored in the Delphi form file thanks to the properties hooking into the reading and writing mechanisms correctly. You can see the Object Inspector showing this in Figure 2.


Object Inspector
ActiveFormX1: TActiveFormX
Properties | Events

Caption	ActiveFormX
Color	clBtnFace
Cursor	crDefault
DragCursor	crDrag
DragMode	dmManual
⊞Font	(TFont)
Height	41
HelpContext	0
Hint	
KeyPreview	False
Left	8
MyNewFormProp	A new property value
Name	ActiveFormX1
ParentColor	True
ParentFont	False

All shown

➤ *Figure 2: A custom ActiveX control property showing in Delphi.*

➤ *Listing 5: Standard ActiveX code that represents a button control.*

```
TButtonX = class(TActiveXControl, IButtonX)
private
  FDelphiControl: TButton;
  ...
end;
...
procedure TButtonX.InitializeControl;
begin
  FDelphiControl := Control as TButton;
  ...
end;
...
initialization
  TActiveXControlFactory.Create(ComServer, TButtonX, TButton, Class_ButtonX,
    2, '', 0, tmApartment);
end.
```

➤ *Listing 6: A new button class with a new published property.*

```
TMyButton = class(TButton)
private
  FMyNewButtonProp: String;
published
  property MyNewButtonProp: String read FMyNewButtonProp write FMyNewButtonProp;
end;
```

➤ *Listing 7: A new button class with a new published property.*

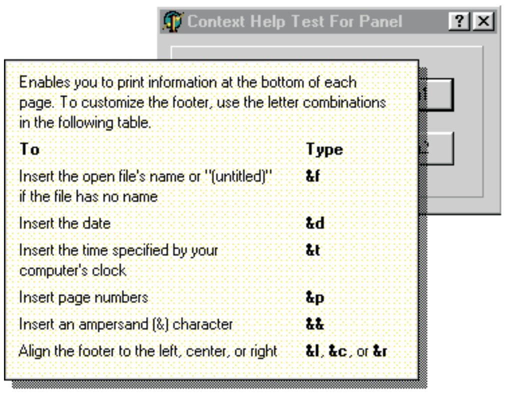
```
TButtonX = class(TActiveXControl, IButtonX)
private
  FDelphiControl: TMyButton;
  ...
protected
  ...
  function Get_MyNewButtonProp: WideString; safecall;
  procedure Set_MyNewButtonProp(const Value: WideString); safecall;
end;
...
procedure TButtonX.InitializeControl;
begin
  FDelphiControl := Control as TMyButton;
  ...
end;
...
function TButtonX.Get_MyNewButtonProp: WideString;
begin
  Result := FDelphiControl.MyNewButtonProp
end;
procedure TButtonX.Set_MyNewButtonProp(const Value: WideString);
begin
  FDelphiControl.MyNewButtonProp := Value
end;
initialization
  TActiveXControlFactory.Create(ComServer, TButtonX,
    TMyButton, Class_ButtonX, 2, '', 0, tmApartment);
end.
```

Unhelpful Panel

Q I am in the process of upgrading a Delphi 2 project to Delphi 5 and I have noticed an inconsistency in the behaviour of one of the standard components. The application uses the context help option (the question mark button on the caption bar that can be used to get context-sensitive help for controls with a non-zero `HelpContext` property).

I use a lot of panels, with various controls on them. The problem is that many of my panels have a non-zero `HelpContext` property value. When compiled under Delphi 2, the context help button works fine for all controls including panels. Having recompiled the project in Delphi 5, the panels no longer react to the context help button. How can I fix this problem?

A This one took some hunting to track down. Eventually I found the change in the VCL source that causes the problem. The change occurred in Delphi 3, which was the first version to break the panel component's context help support.



Enables you to print information at the bottom of each page. To customize the footer, use the letter combinations in the following table.

To	Type
Insert the open file's name or "(untitled)" if the file has no name	**&f**
Insert the date	**&d**
Insert the time specified by your computer's clock	**&t**
Insert page numbers	**&p**
Insert an ampersand (&) character	**&&**
Align the footer to the left, center, or right	**&l**, **&c**, or **&r**

➤ *Figure 3: Context help invoked from a fixed panel.*

It seems that all `TWinControl` descendants have the potential of being a parent to other controls. Programmatically, any component that inherits from `TWinControl` can be assigned to the `Parent` property of any control. In the Form Designer, however, you can only make a control become a child of another control that has the `csAcceptsControls` member in its `ControlStyle` set property. This is why at design-time you can have controls become children of panels and group boxes, but not children of buttons and edit controls.

In the `CreateParams` method of `TWinControl` in Delphi 2, if the control has `csAcceptsControls` in the `ControlStyle` property then the underlying window style has the `WS_CLIPCHILDREN` flag added to it. In Delphi 3 and later, the window style is modified the same way, but additionally the extended window style has the `WS_EX_CONTROLPARENT` flag added to it.

It is this modification to the extended window style that causes the problem. Windows that are described by this flag as being a potential parent of other controls are considered not appropriate for context-sensitive help by Windows (for reasons best known to Microsoft), so all Delphi container controls will suffer this fate (panels, group boxes, tab sheets and so on).

Clearly, to remedy the problem, you must remove this extended window style from the panel component. I see two ways of achieving this. The simplest way is to use the `GetWindowLong` and `SetWindowLong` Win32 API calls. These can extract and modify the extended window style attribute of any window. In the `OnCreate` event handler of each form with problematic panels, you could call a routine like the one shown in Listing 8, which iterates recursively through all the controls on the form, modifying the style of each control which will suffer from the problem.

You can see this code working in the sample project HelpTest.dpr in Figure 3. The project has been associated with NotePad's help file and the panel was given a `HelpContext` property of `1000`. As you can see, this is the context number for the footer help topic.

The other way is to produce a modified component class, inherited from the affected class, which does not make use of that extended style in the first place. Listing 9 shows the class that is defined in the `DCPanel` unit (on the disk). It overrides the `CreateParams` method to remove the offending style flag that was added in by the ancestor `TWinControl` class.

An obvious question to ask at this stage is whether it is safe to remove this flag. Surely the Borland developers chose to add it in for good reason, right? Well, according to the Windows API help file, the `WS_EX_CONTROLPARENT` extended window style allows the user to navigate among the child windows of the control using the `Tab` key.

Thinking this might affect how a panel would operate on a Delphi modal form, where the panel had child controls on it, I tested it before and after removing the style. It worked just as well either way, so it seemed perfectly safe to remove it.

However, checking further with the Windows SDK, it appears that removing the flag may affect the way that `GetNextDlgTabItem` and `GetNextDlgGroupItem` operate. These two APIs are supposed to find the next (or previous) tab stop or control in a group of controls respectively. If either of these APIs encounter a window with the `WS_EX_CONTROLPARENT` style, then that window's children will be recursively searched.

Clearly, if you never use either of these APIs, then there is little to worry about.

➤ *Listing 8: Modifying the extended window style of a panel to enable context help.*

```
// Remove WS_EX_CONTROLPARENT, where set, to allow context help to work
procedure FixContextHelp(Parent: TControl);
var
  I, OldExStyle: Integer;
  Handle: HWnd;
begin
  if (csAcceptsControls in Parent.ControlStyle)
    and (Parent is TWinControl) then begin
    Handle := TWinControl(Parent).Handle;
    OldExStyle := GetWindowLong(Handle, GWL_EXSTYLE);
    SetWindowLong(Handle, GWL_EXSTYLE, OldExStyle and not WS_EX_CONTROLPARENT);
    with TWinControl(Parent) do
      for I := 0 to ControlCount - 1 do
        FixContextHelp(Controls[I]);
  end
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FixContextHelp(Self)
end;
```

➤ *Listing 9: A custom panel component that supports context help.*

```
type
  THelpPanel = class(TPanel)
  protected
    procedure CreateParams(var Params: TCreateParams); override;
  end;
...
procedure THelpPanel.CreateParams(var Params: TCreateParams);
begin
  inherited;
  Params.ExStyle := Params.ExStyle and not WS_EX_CONTROLPARENT;
end;
```

Corrections

Back in Issue 47 (last July), I was discussing the habit of setting object references to `nil` after destroying the objects they refer to. When the object reference is local, this `nil` assignment produces a hint about the assignment of the value not being used. I suggested that this code would disable the hint:

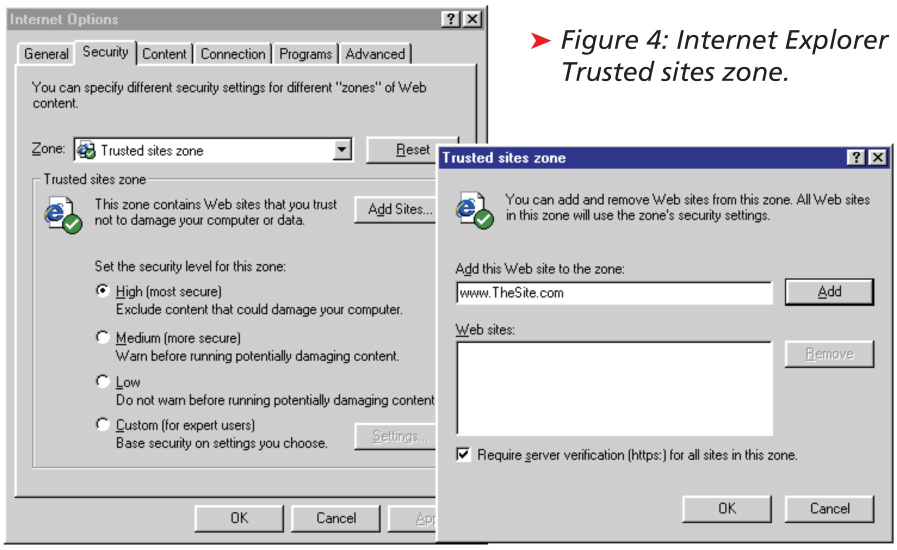
```
{$Hints Off}
  Bmp := nil;
{$Hints On}
```

Unfortunately, as Neil Cullen pointed out to me, this suggestion does not work. The `$Hints` directive is procedure-based. Inserting the directives mid-routine has no effect. To correctly disable this hint you must place the `$Hints Off` directive before the subroutine, and `$Hints On` after the routine.

In Issue 53, the subject of Authenticode was covered and I discussed the possibilities of code signing your ActiveX controls. However, as Kosmas Chatzimichalis told me recently, code signing costs money, which might be beyond the realms of possibility for shareware authors. An alternative scheme is to make use of the `Trusted Sites` zone in Internet Explorer.

This option allows you to add an arbitrary site to a list of sites that are implicitly trusted. In Internet Explorer 4, choose `View | Internet Options...`, then select the `Security` page. From here, drop down the `Zone` combobox and choose the `Trusted Sites zone` option (see Figure 4). You can then use the `Add Sites...` button to add in specific sites to this zone. The security setting for the `Trusted Sites zone` can then be set to `Low`, whilst leaving the setting for `Internet zone` set to `High`.

Thanks are due this month to both Neil and Kosmas.


Internet Options

You can specify different security settings for different "zones" of Web content.

Zone: Trusted sites zone

Trusted sites zone

This zone contains Web sites that you trust not to damage your computer or data.

Set the security level for this zone:

[✓ High (most secure)] Exclude content that could damage your computer.

Medium (more secure) Warn before running potentially damaging content.

Low Do not warn before running potentially damaging content

Custom (for expert users) Base security on settings you choose.

Trusted sites zone

You can add and remove Web sites from this zone. All Web sites in this zone will use the zone's security settings.

Add this Web site to the zone: www.TheSite.com

Web sites:

[✓] Require server verification (https:) for all sites in this zone.

➤ *Figure 4: Internet Explorer Trusted sites zone.*