Delphi Meets COM: Part 6

by Dave Jewell

The best laid plans of mice and men don't always work out as one would hope, and the same is certainly true of programmers and computer journalists! At the end of last month's instalment, I mentioned my plans to do something naughty with the Delphi IDE. In essence, I intended to convert the IDE into an OLE Automation server. The idea was that a client application would be able to drive the IDE on 'auto-pilot', so to speak, accessing many of the Open Tools goodies that we've come to know and love. Well, that was the idea, and things were looking good until I discovered that the COMSERV unit (which is rather vital for implementing Automation servers) can't be compiled into a package. Oh well, it seemed like a good idea at the time...

I'm still pondering this and may well come back to it, but in the meantime let's move swiftly on and take a look at ActiveX controls. As I may have mentioned before, an ActiveX control (OCX) is essentially a specialised in-process COM server. It lives in the same address space as the container application and communicates with the container through a number of specialised interfaces. In Delphi, ActiveX controls are descendants of the TActiveXControl class.

As you'll no doubt appreciate, the simplest way of creating an ActiveX control is to convert an existing VCL component into an OCX using Borland's 'One Touch' ActiveX technology. In practice, you'll often find that more than one 'touch' is required! This is because, as I mentioned last month, only certain data types are automation-compatible, so a converted VCL may have certain properties missing when it comes out of the ActiveX sausage machine.

Introducing TDesktop

Let's put all this into concrete terms by taking a VCL component and putting it through the ActiveX conversion process. Rather than using an existing VCL component, which would be cheating, I decided to create a sample component from scratch. The source code to this, TDesktop, is given in Listing 1. This is a 'for fun only' component that doesn't do anything terribly useful: it's for illustrative purposes only. The only purpose of TDesktop is to allow you to programmatically modify certain display attributes of the Windows desktop, in particular the colour and the font used to display the text under the icons.

Although you may not be aware of it, the Windows desktop is actually implemented using a standard listview control (I'm obviously talking about the low-level API implementation of the listview control, not the VCL wrapper for the listview control which sits on Delphi's component palette). Microsoft also used the listview control inside Windows Explorer: you can see it being used in the right hand Explorer pane. It's pretty obvious that Explorer is using listview controls, but things are much less obvious in the case of the desktop. There's no control 'background' and each icon seems to have a life of its own. Nevertheless, the entire desktop is indeed managed by one control; the key question is how we get access to it?

The technique I present here is not my own. I first saw it used in one of the sample programs that comes with the pre-release version of Visual J++ 6.0 *[Don't miss Dave's review of this in June's* ***Developers Review****! Ed]*, converted it into Delphi code and have unashamedly incorporated it into the TDesktop component.

The relevant code appears in the constructor method of TDesktop. Here, a handle to the desktop window is first retrieved through a call to GetDesktopWindow. Then, the FindWindowEx routine is used to enumerate the child windows of the desktop, looking for a window with class Progman and name Program Manager. This identifies the hidden shell itself, which you shouldn't confuse with the old Windows 3.1 Program Manager. The reason why the shell window uses these names is for backward compatibility: some old installer programs look for a window of this name and class and then direct DDE messages to the window in order to create program manager groups. Under Windows 95, these DDE messages are intercepted and the equivalent Start menu entries are created instead.

Next, the code looks for a child window of the Progman window, searching for a class name of SHELLDLL_DefView. This corresponds to the default view of the Windows Explorer. Finally, the children of this window are enumerated in order to find a control which has the class SysListView32. If you're familiar with the underlying common controls DLL, you'll know that this is the API-level class name of the listview control.

Once we've got a handle to this control, we can do all sorts of interesting things with it, as defined in the COMMCTRL.PAS file. In that unit, you'll find a number of lvm_xxxx constants which correspond to different messages that can be sent to the control. As a simple example, you'll see that the TDesktop component implements a read-only property called ItemCount. This returns the total number of items present on the Windows desktop. This property is implemented through the private GetItemCount method which, in turn, calls the ListView_GetItemCount routine defined in COMMCTRL. This routine sends a lvm_GetItemCount message to the listview control using the SendMessage API call.

In the same way, there's a TextColor and a TextBackgroundColor property which provide access to the icon text colour (and background colour) respectively. Again, these properties work by sending lvm_xxxx messages to the desktop listview window in order to set and query attributes of the control. One interesting wrinkle here is the listview control's ability to display icon text with a transparent background. In other words, display text where the window background shows through. This is done by passing an invalid colour value of $ffffffff to the listview control, using the lvm_SetTextBkColor message. Rather than relying on the developer understanding the special significance of this 'colour value' I decided to 'overload' the meaning of the predefined colour value clNone. Thus, when the SetTextBackgroundColor routine detects that a value of clNone is being passed, it converts it to the 'magic' value before passing to the listview window. This is an interesting example of achieving some effect through API-level manipulation that you couldn't duplicate if, for example, you were simply working with the VCL listview 'wrapper'. See Figure 1 for an example of what the effect looks like, with a suitably scenic excerpt from my desktop!

Finally, the TDesktop control also exposes a Font property which, as you'd expect, can be used to directly alter the font used for displaying icon titles. All sorts of interesting psychedelic effects can be achieved by using over-large, decorative fonts for your Windows desktop! The SetFont method works by passing the selected font to the control with the standard Windows wm_SetFont message, a technique which works with all the Windows common controls.

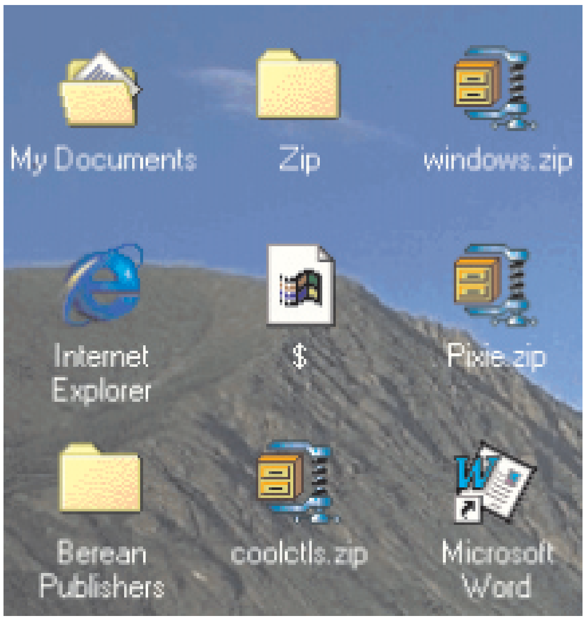
Once again, I must emphasise that this component is provided for illustrative purposes only. In a serious TDesktop component, you'd want to change the persistent state of the desktop rather than simply alter the desktop appearance on the fly. As it is, no matter what wacky things you do to your desktop, sanity will be restored when you restart Windows, which is probably just as well!

TComponent? Just Say No...

You may already have noticed one odd thing about the TDesktop component. Although this is, to all intents and purposes, a non-visible component, I've derived it from TCustomControl instead of from TComponent. What's the reason for this? I specifically did things this way because Delphi's ActiveX control wizard will ignore non-visible

➤ *Listing 1*

```
unit Desktop;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs;
type
  TDesktop = class (TCustomControl)
  private
    fListView: hWnd;
    fFont: TFont;
    function GetItemCount: Integer;
    procedure SetItemCount (Value: Integer);
    procedure SetTextColor (Value: TColor);
    function GetTextColor: TColor;
    procedure SetTextBackgroundColor (Value: TColor);
    function GetTextBackgroundColor: TColor;
    procedure SetFont (Value: TFont);
  protected
    procedure Paint; override;
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Font: TFont read fFont write SetFont;
    property ItemCount: Integer read GetItemCount
      write SetItemCount;
    property TextColor: TColor read GetTextColor
      write SetTextColor;
    property TextBackgroundColor: Tcolor
      read GetTextBackgroundColor
      write SetTextBackgroundColor;
  end;
procedure Register;
implementation
uses CommCtrl;
constructor TDesktop.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  Width := GetSystemMetrics (sm_CxIcon);
  Height := GetSystemMetrics (sm_CyIcon);
  Visible := False;
  fFont := TFont.Create;
  // Start with the desktop window
  fListView := GetDesktopWindow;
  // Search through desktop children for the shell
  fListView := FindWindowEx (fListView, 0, 'Progman',
    'Program Manager');
  // Search shell's children for the shell default view
  fListView := FindWindowEx (fListView, 0,
    'SHELLDLL_DefView', Nil);
  // Finally, get the listview that's inside the shell
  // default view
  fListView :=
    FindWindowEx (fListView, 0, 'SysListView32', Nil);
end;
destructor TDesktop.Destroy;
begin
  fFont.Free;
  Inherited;
end;
procedure TDesktop.Paint;
begin
  if csDesigning in ComponentState then
    DrawIcon(Canvas.Handle, 0, 0, LoadIcon(0, idi_WinLogo));
end;
procedure TDesktop.SetFont (Value: TFont);
begin
  fFont.Assign (Value);
  SendMessage(fListView, wm_SetFont, fFont.Handle,
    Integer(True));
end;
procedure TDesktop.SetTextColor (Value: TColor);
begin
  Value := ColorToRGB (Value);
  if Value <> GetTextColor then begin
    ListView_SetTextColor (fListView, Value);
    InvalidateRect (fListView, Nil, True);
  end;
end;
function TDesktop.GetTextColor: TColor;
begin
  Result := ListView_GetTextColor (fListView);
end;
procedure TDesktop.SetTextBackgroundColor (Value: TColor);
begin
  // If wanted color is clNone, interpret as Transparent
  if Value = clNone then Value := $ffffffff else
    Value := ColorToRGB (Value);
  if Value <> GetTextBackgroundColor then begin
    ListView_SetTextBkColor (fListView, Value);
    InvalidateRect (fListView, Nil, True);
  end;
end;
function TDesktop.GetTextBackgroundColor: TColor;
begin
  Result := ListView_GetTextBkColor (fListView);
end;
function TDesktop.GetItemCount: Integer;
begin
  Result := ListView_GetItemCount (fListView);
end;
procedure TDesktop.SetItemCount (Value: Integer);
begin
  // Read only property
end;
procedure Register;
begin
  RegisterComponents ('Samples', [TDesktop]);
end;
end.
```



➤ *Figure 1: The hills are alive... with the sound of ActiveX control development! Here's a close up of what your desktop will look like if you use the undocumented listview control feature whereby it's possible to set a transparent background colour for the icon text*

components when giving you a list of possible VCL classes from which to derive your new ActiveX component. Personally, I think this is a rather irritating state of affairs, but doubtless there's some legitimate reason hidden inside Borland's ActiveX control framework.

Anyway, because I derived from `TCustomControl`, I also added a little bit of code to give an initial height and width to the control, and a `Paint` procedure so that the Windows logo icon would be used to identify the control when using it on a design-time form. With all this in place, we're ready to convert our new control into an ActiveX component and see what happens.

To do this, just close your existing project (if any), open a new ActiveX library project, save it, and then create a new ActiveX control using the control wizard, (see Figure 2). Once you click `OK`, Delphi will generate a number of source files and a type library for the control. If you then `Build` the project, you'll end up with a 284Kb OCX control. This reduces to an impressively sylph-like 31Kb if you build using runtime packages.

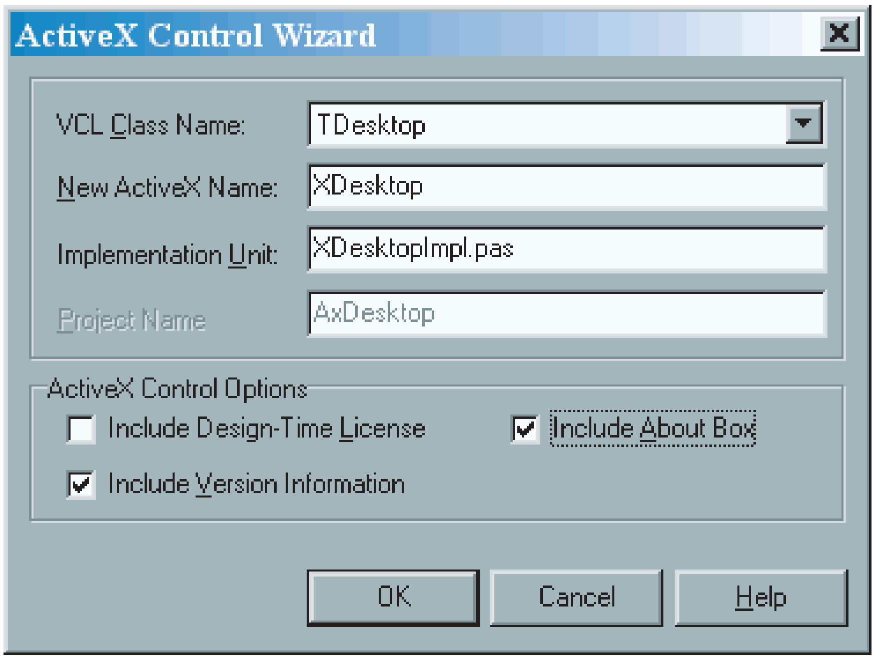
Incidentally, referring back to Figure 2, you'll notice that there are three checkboxes associated with the ActiveX control wizard. The first, `Include Design-Time License`, is used to ensure that a control will not operate in design mode unless an appropriate license file (with the extension .LIC) is present. This prevents unscrupulous developers from incorporating your control into a new program after having discovered it as part of the runtime support of some other application. If you click this checkbox, the ActiveX control wizard will automatically generate an appropriate .LIC file entry and will add the necessary checking logic to the OCX file itself.

You'll also see another checkbox marked `Include About Box`. If you check this box Delphi will include another form into your OCX project. This form implements an About box for the control and is called when (for example) the developer clicks your control's `About...` property in Visual Basic's property inspector.

Finally, there's another checkbox marked `Include Version Information`. If checked, this will include a standard Windows `VERSION` resource into your OCX file. According to the Borland documentation, Visual Basic 4.0 will refuse to load an OCX control unless it contains version information, so it's a good idea to always check this box. Just as importantly, by including a version resource in your ActiveX control, it makes it possible to expose version numbers and copyright notices to developers who use your control.

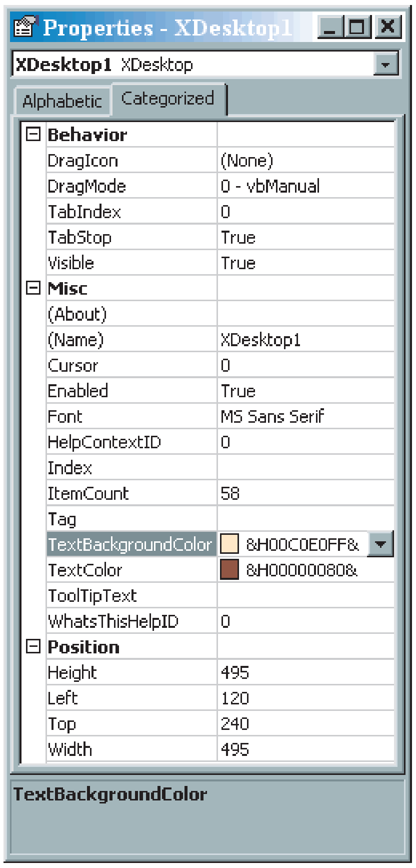
Once you've built the OCX control, the next step is to register it. You can do this without leaving the IDE by choosing the `Register ActiveX Server` menu item from the `Run` menu. When you do this, the Delphi IDE registers the control just as it would be registered on the end-user's machine. In other

➤ *Figure 2: The ActiveX control wizard has a deceptively simple-looking interface, but its implementation requires a large amount of code in the Delphi IDE. Here, we've specified that our control should have an About box, and we've indicated that the IDE should include a standard Windows VERSION resource in the OCX file*


ActiveX Control Wizard

VCL Class Name: TDesktop
New ActiveX Name: XDesktop
Implementation Unit: XDesktopImpl.pas
Project Name: AxDesktop

ActiveX Control Options
[ ] Include Design-Time License
[✓] Include About Box
[✓] Include Version Information



Properties - XDesktop1

XDesktop1 XDesktop

Alphabetic | Categorized

⊟ **Behavior**	
DragIcon	(None)
DragMode	0 - vbManual
TabIndex	0
TabStop	True
Visible	True
⊟ **Misc**	
(About)	
(Name)	XDesktop1
Cursor	0
Enabled	True
Font	MS Sans Serif
HelpContextID	0
Index	
ItemCount	58
Tag	
TextBackgroundColor	&H00C0E0FF&
TextColor	&H00000080&
ToolTipText	
WhatsThisHelpID	0
⊟ **Position**	
Height	495
Left	120
Top	240
Width	495

TextBackgroundColor

➤ *Figure 3: Here's our TDesktop control, as seen through the eye of the Visual Basic 5.0 Properties window. This is the state of play immediately after running ActiveX control wizard and building the OCX*

words, the IDE looks for a routine inside the OCX called `DllRegisterServer`, and calls it. This has the effect of registering the OCX control with the system registry.

At this point, you can fire up another development system, such as Visual Basic, and try out the `TDesktop` control, now transmogrified into the `AxDesktop` control. Assuming you're working with VB 5, launch the development system, create a new EXE-style project and right click on the VB component palette. Click the `Components...` entry in the context menu and you'll find yourself in the `Components` dialog. This will give you a list of all registered OCX controls which should include, all being well, an entry called `AxDesktop Library`. If you click this item and press the `OK` button, you'll see that an `XDesktop` control has appeared on the palette. You can now manipulate the `TextColor` and `TextBackgroundColor` properties just as you did in the original VCL control, and you'll see that the Windows desktop dutifully redraws itself to match the property changes that are made. Success!

Conversion Issues: Time For A Massage?

Before you rush out and tell all your friends that you're a real whizz at creating ActiveX controls, you need to take a long hard look at `XDesktop` as seen from the Visual Basic Property Inspector, (Figure 3). For example, you'll notice that the `Visible` property of the control defaults to `True`, which isn't what we want. There's also an `Enabled` property, a `Cursor` property and other things that we'd prefer to eliminate. The `ItemCount` property seems to have become read/write (although, mercifully, assigning a value to it is ignored!) and changes to the `Font` property are politely ignored. Obviously, a certain amount of massaging is called for!

If you've had much experience of writing Delphi components, you'll know that it isn't possible to make a property declaration less visible in a derived class than it already was in the ancestor class. In concrete terms, we can't go back to the source code of `TDesktop`, incorporate a property declaration for the `Cursor` property (for example) into the private part of the class and expect the `Cursor` property to become invisible. The code will compile, but things will look just the same in Delphi's Object Inspector and in Visual Basic's Properties window. Obviously, some other approach is needed.

The key, of course, is the type library. If you open up the ActiveX library project which contains the `XDesktop` control, you should then be able to access the type library from the `View|Type Library` menu option. Once the type library editor has appeared, open the `IXDesktop` interface in the editor and delete the `Cursor` property. If the `XDesktopImpl` unit isn't currently loaded into the IDE, the type library editor will ask if you want to load it; say `Yes`. If you now rebuild the OCX and re-launch VB, you'll find that the `Cursor` property has disappeared from the `Properties` window. Suitably flushed with success, you can do the same thing with the `Enabled` property.

That's not quite the end of the story, though. If you examine the definition of `TXDesktop` in the `XDesktopImpl` unit, you'll find that the `Get_Cursor` and `Set_Cursor` methods are still defined and weren't automatically deleted by the type library editor. You can safely delete them yourself. The same is true for the `Set_Enabled` and `Get_Enabled` routines.

Next, take a look at the `ItemCount` property in the type library. You'll see that it's defined as being a read/write property, whereas we would ideally like it to be read-only. It's not surprising that the ActiveX control wizard made this a read/write property because I defined both 'get' and 'set' routines in the original VCL control. I did this because, as you probably know, the Delphi Object Inspector won't display read-only properties. You have to fool it into thinking that it's dealing with a read/write property by providing a dummy 'set' routine.

You might imagine that you can convert `ItemCount` into a read-only property by just selecting the `ItemCount` property in the type library editor and selecting read-only from the associated property type combobox. But if you do that, and then rebuild the OCX, you will then find that `ItemCount` has disappeared from Visual Basic's Properties window!

Maybe you're thinking that `ItemCount` isn't a property any more? As with Delphi, Visual Basic has a code completion facility and even though `ItemCount` is no longer in the Properties window, you can verify that it's still 'visible' to VB by typing something like:

```
wombat = XDesktop1.ItemCount
```

As soon as you've typed the period after the component name, VB will display a list of possible properties which will include `ItemCount`. So what's going on? The fact is, VB's

Properties window is just as coy about displaying read-only properties as Delphi's Object Inspector! The rule seems to be, 'if I can't tweak it, I'm not going to show it.'

Given that VB won't display read-only properties in the Properties window, this raises the question of whether it's worth tweaking the type library at all in this specific case: should we rely on the fact that the underlying VCL component will effectively ignore assignments to this property, or should we mark all read-only properties as such in the type library? This is really a matter of personal choice: if you leave the property as notionally read-write, it will appear in property browsers, but the developer who doesn't read the documentation will be confused when his or her property assignment is ignored. If you set the property as read-only, it won't appear in property browsers and the development system will itself flag any attempt to assign to the property. In the case of VB 5.0, you'll get an error number 450 (*Wrong number of arguments or invalid property assignment*) when the interpreter tries to execute an assignment to a read-only property.

On a somewhat different tack, you'll notice that VB's Properties window has a small area at the bottom which provides hint information on the selected property. This provides instant feedback to the programmer on what the property does. If you want to provide this sort of information to users of your control, it's very easy to do: just use the type library editor to open the `IXDesktop` interface and type an entry into the `Help String` field for every property, event and method that's exposed by the interface. Rebuild the control, move back to VB, and all the hint strings that you've specified will appear in the Properties window. For a complex OCX with many properties, events and methods, this simple step greatly increases the usability of the control.

Help With Help

While on the subject of providing help for the developer who uses your control, you should ideally include a help file which gives more detailed information on how to use the control and documents each of the exposed properties, events and methods. Back in the Delphi type library editor, select the topmost node of the open type library tree (for want of a better name, the odd-looking group of three squares at the extreme top of the content pane). Once you've done that, the Attributes pane will allow you to set 'global' attributes which relate to the entire type library. This includes the name of the help file associated with your control. Enter the name of your control's help file here, then rebuild the OCX and load the control under Visual Basic. Now, you'll find that if you put the control on a form and (with the control selected) press the `F1` key, Visual Basic will bring up the help file you specified. If you want to designate a particular topic as the introductory help topic for that control, you can specify that topic's help context number in the global attributes pane for the type library. Finally, to set an individual help context number for each property, method and event, use the help context edit box in the attributes pane that's displayed when a property, method or event is selected.

Assuming that you told the ActiveX control wizard you wanted an About box, there will already be a standard Delphi form as part of your control project. You set this up in the way you want just as you'd set up any other Delphi form. You will almost certainly want to do this because, as the control wizard creates it, the copyright notice states *Copyright © 1997 Frank Borland*! And if you're wondering who Frank Borland is, suffice it to say that Uncle Sam is to America as Frank Borland is to Borland (if you have an old version of Borland's Resource Workshop, you might be able to see Frank's hat hiding in the About box...).

Incidentally, concerning the way in which the control wizard creates the About box, you'll notice that the `About1` unit (or whatever it happens to be called) effectively lives in a world of its own. In other words, it exports a single entry point, `ShowXDesktopAbout` (which displays the About form), and doesn't use any of the COM/ActiveX related units. This means that you can't easily 'get at' important information, such as the version number etc, from the About box. The simplest way to rectify this is to add the `ComServ` unit to the uses clause of the About form unit. You've then got access to the various goodies defined by the `ComServer` variable.

Conclusion

This month has served as a gentle, tutorial-based, introduction to Delphi ActiveX control development, the aim being to make you feel comfortable with the basics of ActiveX controls. Next month we'll be looking in more detail at the code generated by the ActiveX control wizard, examining some of the more important classes in DAX (the Delphi ActiveX Framework, which is responsible for implementing ActiveX controls within the Delphi environment) and the ways in which DAX greatly simplifies ActiveX control creation.

The source code to the `TDesktop` VCL component, together with the ActiveX control project, is included on this month's disk. I've included a pre-built version of the OCX, but please note that, in order to save disk space, this is the 'packaged' version and requires the Delphi 3 runtime packages. Also bear in mind that I've only tried hosting this control under Visual Basic 5.0: if you encounter problems with hosting it on other development systems, then you're on your own!

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave as Dave@HexManiac.com