The Delphi CLINIC

Edited by Brian Long

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

Extending Actions

Q I am using a standard action (such as `TDataSetFirst`), but want to make an `OnExecute` event handler for it to extend its behaviour. However, the act of making an `OnExecute` event handler stops the pre-defined standard action behaviour from occurring. How do I overcome this problem?

A It seems that we could answer two action-related questions here. One would apply to custom (user-defined) actions, the other to standard (Borland-defined) actions.

The first question would be: 'If I set up an action and connect it to a button, the button's `OnClick` event shares the action's `OnExecute` event handler. How can I extend the behaviour of the button so it performs the action's code, and also runs some extra code of my choosing?' The second question would be the original one above, regarding extending the behaviour of a standard action.

Let's take the custom action question first, and deal with the original question afterwards. As mentioned, the usual behaviour with custom actions is to define their behaviour in their `OnExecute` event handler. When you then connect the action to an action client, such as a button (using its `Action` property), the button's `OnClick` event shares the action's `OnExecute` event handler.

In order to extend the behaviour of the button, the idea would be to clear the button's `OnClick` event, thereby stopping it sharing the action's `OnExecute` event handler. You then make a normal `OnClick` event handler for the button and add in the extra code there. The problem occurs when trying to trigger the action's code from the new event handler. This is achieved by calling the action's `Execute` method at an appropriate point in the `OnClick` event handler.

That's it for extending the behaviour of an action client connected to a custom action; fairly straightforward I'm sure you'd agree. Let's now look at the original question, which refers to standard actions.

A standard action has all of its functionality built in, and so does not require an `OnExecute` event handler to do its job. In fact, as the questioner points out, the very act of making an `OnExecute` event handler stops the standard action doing its job. So we need to find out how to invoke the action's built-in behaviour from within the `OnExecute` event handler.

Unfortunately, it is not as simple as it was with the first question. Calling the action's `Execute` method will cause an infinite loop, since it will just call the `OnExecute` event handler, which will call `Execute`, which will call the `OnExecute` event handler, and so on. Instead, we have to take account of how standard actions work in the first place.

You can read all the gory details in my *Actions And Action Lists* article from Issue 61 (September 2000), but the short story is that they work on the assumption that there is no `OnExecute` event handler. After finding no `OnExecute` event handler, the application takes another approach for standard actions.

Let's focus on standard actions that operate on the active control, such as `TEditCopy` and `TEditPaste`. When the keystroke for such an action is pressed (remember that standard actions do not have to be connected to any action clients), or the menu item is selected, the application asks the action if it can work against the active control. If the action says it cannot, the application offers it to each visible control on the active form, then each visible control on the main form. When the action is given a control it says that it can work against (the first edit control passed to it) it is then asked to do its job against that control.

The `HandlesTarget` method is used to find out if the action can operate against some control and `ExecuteTarget` is called to make the action do its job against that control. If you know which control should be used as the action client (such as the one referred to by the form's `ActiveControl` property, or maybe just some specific control of your choice) you can make the `OnExecute` event handler look like Listing 1.

However, if you are not concerned with which component will be affected by the action, then you can generalise the code a bit and send the same message to the `Application` object that an action would normally send when it finds it has no `OnExecute` event handler. This is shown in Listing 2.

Note that `SendAppMessage` is not in the Delphi 5 help, but is a shorthand way of sending a message to

➤ *Listing 1: Extending the behaviour of an action, when the action client is known.*

```
procedure TForm1.EditCopy1Execute(Sender: TObject);
begin
  Caption := Format('The %s action struck at: %s',
    [Sender.ClassName, TimeToStr(Time)]);
  with Sender as TAction do
    if HandlesTarget(ActiveControl) then
      ExecuteTarget(ActiveControl)
end;
```

the `Application` object, without checking that it has an underlying window handle in advance.

The questioner specifically mentioned a dataset standard action, and it should be mentioned that these standard actions act on a `TDataSource` component, which is not a control. Consequently, if you weren't using the approach shown in Listing 2, it would be necessary to pass a `TDataSource` component to both `HandlesTarget` and `ExecuteTarget`, or even the dataset action's own `DataSource` property, if it has been set (see Listing 3).

```
procedure TForm1.EditCopy1Execute(Sender: TObject);
begin
  Caption := Format('The %s action struck at: %s',
    [Sender.ClassName, TimeToStr(Time)]);
  SendAppMessage(CM_ACTIONEXECUTE, 0, Longint(Sender))
end;
```

➤ *Listing 2: Extending the behaviour of an action, regardless of the action client.*

```
procedure TForm1.DataSetPost1Execute(Sender: TObject);
begin
  Caption := Format('The %s action struck at: %s',
    [Sender.ClassName, TimeToStr(Time)]);
  with Sender as TDataSetAction do
    if HandlesTarget(DataSource) then
      ExecuteTarget(DataSource)
end;
```

➤ *Listing 3: Extending a dataset action.*

InActiveX

Q I am developing an ActiveX in Delphi and already have a number of them under my belt. This particular one is causing me a problem when being referenced from an HTML page. No matter what I try, I cannot get the ActiveX to be displayed in Internet Explorer. It doesn't even give me a red cross ( ) where the ActiveX should be, which might indicate an error. It just leaves the original ActiveX placeholder image ( ).

The really confusing thing is that I can import the ActiveX into Delphi and drop an instance of it on a form just fine. What could be wrong with the control?

A I've seen cases where it was difficult to get *any* ActiveX controls to show up in Internet Explorer, for various reasons, but this is a problem with one particular control. With the facts as they have been laid out so far, I would guess this is a licensing issue. To see how valid my guess was, I did some testing and came up with exactly the same behaviour described in the question. Allow me to elucidate.

To reproduce this problem, start Delphi and ask for a new `ActiveX Library` from the `ActiveX` page of the `File | New...` dialog. Save this project under whatever name you choose (for example, MyActiveXLib.dpr). Now create an ActiveX by choosing `ActiveX Control` from the same page of the `File | New...` dialog, which invokes the ActiveX Control Wizard.

For this simple example, we will just choose a `TButton` to base the ActiveX on, so fill in the dialog as shown in Figure 1 (changing the file name as you like). Note the fact that the `Make Control Licensed` checkbox is checked.

There are three consequences of setting this checkbox. The first is that the ActiveX coclass will have the `Licensed` flag set in the type library. This tells COM that it must use the `IClassFactory2` interface to instantiate the COM object, rather than the usual `IClassFactory`. Note that `IClassFactory2` has extra methods which are used to manage licence information (`GetLicInfo`, `RequestLicKey` and `CreateInstanceLic`).

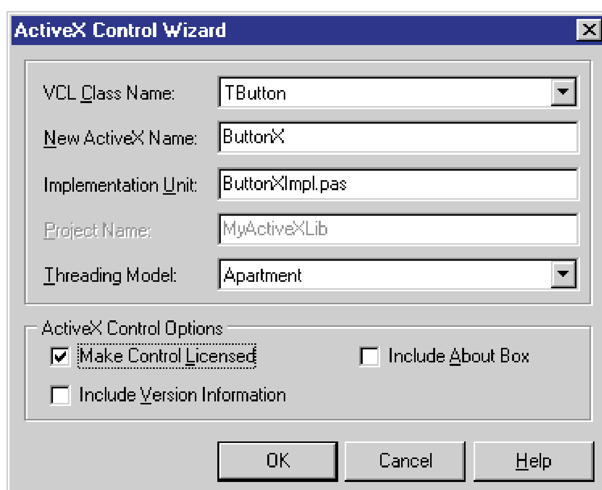
The next thing that happens is that a licence file is generated on the hard disk in the same directory as the generated ActiveX source file. The licence file has the same name as the ActiveX library, but with a .LIC extension. It is a text file that contains a licence key for each licensed ActiveX in the project, one line for each. The licence key can be any string, and the ActiveX Control Wizard chooses to use a new GUID for the purpose.

If the licence file already exists, because the ActiveX library previously contained one or more licensed controls, the new licence key is added as a new line at the end of the file.

The third effect of setting the licence checkbox is that the licence key is passed to the `LicStr` parameter of the ActiveX's class factory constructor, in the ActiveX control unit initialisation section.

The general behaviour of a container program using the ActiveX (such as a development tool) is that the licence information will be verified before it can be used. This is typically done by the ActiveX class factory analysing the .LIC file and checking for the required licence key information.

COM will call the class factory's `CreateInstanceLic` method (from the `IClassFactory2` interface) to create an instance of the ActiveX control (as opposed to `CreateInstance`, from `IClassFactory`,



➤ *Figure 1: The ActiveX Control Wizard.*

which is called for a non-licensed control).

`CreateInstanceLic` normally checks the .LIC file to see if the licence information is present and correct (remember the correct licence information is passed to the class factory constructor). The questioner finds that he can use the problematic ActiveX in Delphi without problem. This is because the .LIC file is sitting in the same directory as the generated ActiveX.

A development tool can query an ActiveX for its licence key (by querying the ActiveX through the `RequestLicKey` method) and the key can be compiled into any application using the ActiveX. By doing this, the runtime application does not need the .LIC files to be present.

A web page with an ActiveX embedded in it can be considered a developed application. However, embedding the licence key in the HTML is not an option, as any web surfer user can read the HTML source any time they like. Instead, Internet Explorer expects to find a reference to a licence package to be embedded in the HTML, describing the licence information for all licenced controls referenced on that page.

A licence package is a file with a .LPK extension and is generated with the License Package Authoring Tool, LPK_Tool.exe. This tool can be downloaded from Microsoft's website:

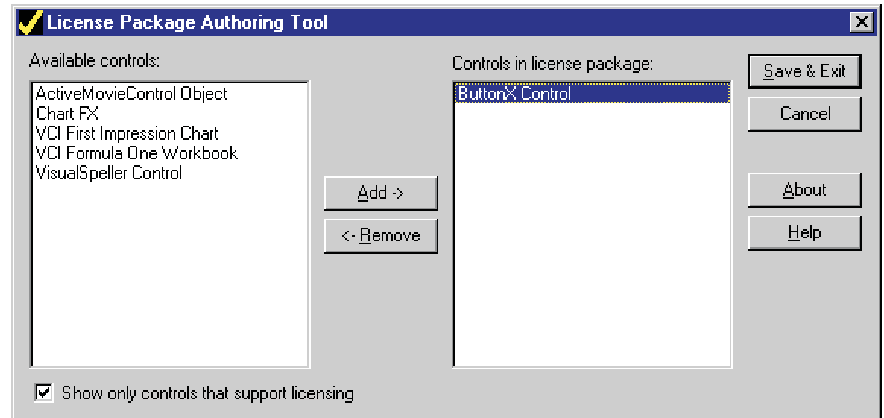
```
http://msdn.microsoft.com/
  downloads/tools/lpktool/
  lpktool.asp
```

or found in the Platform SDK.

Given this information, you might expect to get some form of error message from Internet Explorer if it fails to find a required licence package for an ActiveX that supports licensing. However, I found that if its security settings are set to `Low` I get the same lack of error message as described in the question (I tested with Internet Explorer 4).

Having compiled your ActiveX control, and registered it on your development machine, you can run the License Package Authoring Tool. This shows you a list of all registered ActiveX controls and allows you to choose which ones to place references to in the generated licence package. A checkbox allows you to only see those controls that support licencing.

As far as LPK_Tool is concerned, if the control supports `IClassFactory2`, then it supports licencing. Since all Delphi-generated ActiveX controls implement `IClassFactory2`, then you get all the registered, Delphi-generated ActiveX controls in the list, regardless of whether they actually require licence keys or not. So you will still have to look carefully if you only want appropriate controls referenced in the licence package file.



➤ *Figure 2: LPK_Tool.exe in action.*

Once you have selected the controls to add to the package, you press the `Add` button to move those control names to a separate list (see Figure 2). Finally the `Save & Exit` button lets you choose a file name and saves the licence package file.

An .LPK file is a text file containing a header, a copyright notice (to dissuade casual copying of the file), a GUID that identifies the LPK file format version and then all the licence information. This information is stored as uuencoded information (to make it textual) containing the licence details for each control. An example file is shown in Listing 4.

Having generated a licence package file, the next job is to insert a reference to it in your HTML file. This is done using an `OBJECT` HTML tag, as shown in Listing 5. The class ID is that of the Microsoft Licensed Class Manager, which should be installed on your system. It takes a parameter called `LPKPath` which specifies the relative location of the licence package, with respect to the HTML page.

There can be references to several licence packages in an HTML page, but Internet Explorer ignores all but the first one. When it needs to display a licenced ActiveX, Internet Explorer parses the licence package file, extracts the appropriate licence key and passes it to the `CreateInstanceLic` method of the ActiveX class factory. If it is accepted, the ActiveX

➤ *Listing 4: A licence package file.*

```
LPK License Package
//////////////////////////////////////////////////////////////////////////////
//  WARNING:  The information in this file is protected by copyright law    //
//  and international treaty provisions. Unauthorized reproduction or       //
//  distribution of this file, or any portion of it, may result in severe   //
//  criminal and civil penalties, and will be prosecuted to the maximum     //
//  extent possible under the law.  Further, you may not reverse engineer,  //
//  decompile, or disassemble the file.                                     //
//////////////////////////////////////////////////////////////////////////////
{3d25aba1-caec-11cf-b34a-00aa00a28331}
CZA0y2Cu1BGW7dSU+sABkA=

AQAAAA=

hQK6fmCu1BGW7dSU+sABkCYAAAB
7ADcARQBCAEEAMAAyADgAQQAtAEEARQA2ADAALQAxADEARAA0AC0AOQA2AEUARAAt
AEQANAA5ADQARgBBAEMAMAAwADEAOQAwAHOA=
```

```
<OBJECT classid="clsid:5220cb21-c88d-11cf-b347-00aa00a28331">
  <PARAM NAME="LPKPath" VALUE="ButtonXControl.lpk">
</OBJECT>
```

➤ *Listing 5: Referencing a licence package file.*

```
uses
  //Add extra units here
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
```

➤ *Listing 6: The default uses clause of a form unit.*

control is then rendered on the page.

The solution to the original problem is therefore to make a licence package for the ActiveX and insert a reference to it in the HTML page.

Unit Usage

Q If I remove components from a form, should I worry about the units that are left in the interface section's `uses` clause that are no longer strictly needed?

A Generally speaking, having a few units in any `uses` clause that are not really needed is not much of a big deal. The only real disadvantage is that if some of those units have initialisation sections (and maybe finalisation sections too), then the code in those sections will be compiled into the program and execute unnecessarily.

Of course, there is also the possibility of some of those units being deleted, thereby causing compiler errors when they cannot be found. However, within the confines of the question, this is only likely to happen if the unit implements a third-party component which is uninstalled.

If you want to try and keep your form unit interface section `uses` clauses as tidy as possible, you can follow these guidelines.

Firstly, wherever possible, you should try and place as many unit references into the implementation section `uses` clause. This should always be possible when the reason for adding the unit to a `uses` clause is because of a reference to a routine or a data type being used in the implementation section of your unit. This will keep the interface section `uses` clause smaller, and will consequently help make compilations more efficient. When one unit uses another unit, the compiler first checks out that unit's interface section, including the units used by the interface section `uses` clause. The fewer units in that `uses` clause, the quicker the compiler can do its job.

Secondly, in those cases where you do need to add units into the interface section `uses` clause, add them at the beginning, rather than the end (see Listing 6).

Any time you want to have the interface section `uses` clause spring-cleaned, delete all the units added after the `Dialogs` unit. `Dialogs` is the last unit in the `uses` clause of a fresh form unit in Delphi 5 (see Listing 6). By doing this, you will be deleting a number of necessary units and (hopefully) some unnecessary units. To have all the needed units added back in, simply save or compile the project. The IDE will refresh the `uses` clause, laying it out in a nice and tidy fashion.

Paper Orientation

Q I am trying to change the default printer's paper orientation and am not getting very far. I cannot find much information on the `DEVMODE` structure that seems to lie at the heart of this type of operation. Any ideas?

A That depends on whether you mean changing the settings for the default printer for the purposes of your application (changing the local printer settings), or for all applications (changing the global printer settings). Let's see what we can find out for both cases.

The normal case would be changing options that only affect your application. Usually, the user takes charge of this, thanks to you providing a printer setup dialog in the program, which is quite straightforward. However, it is not so obvious what to do if you want to make the change programmatically.

As the questioner suggests, the `DEVMODE` record is key to this process. You can find information on this structure either in Microsoft's Platform SDK documentation (which you can find in the Microsoft Developer Network Library CD, for example) or in the Windows SDK help file supplied with Delphi.

`DEVMODE` is a record that contains information about the device initialisation and environment of a printer. The record as defined has a whole variety of fields, but many printer drivers store additional private information at the end of the record. Consequently, you often need to take great care when declaring `DEVMODE` records and passing them to a printer driver, as you may be omitting a variety of necessary pieces of information.

You can get a `DEVMODE` record describing the current capabilities and settings of a printer, and then modify it as you need. One important field in the `DEVMODE` record is `dmFields`, which contains a bitmask of flags indicating which other fields in the record are valid. The flag we are looking for that indicates the paper orientation can be changed is `DM_ORIENTATION`. Assuming this flag is present, we can specify a value of `DMORIENT_PORTRAIT` or `DMORIENT_LANDSCAPE` in the `dmOrientation` field.

There are two remaining questions: firstly, how to get the printer's current `DEVMODE`, and secondly how to update the `DEVMODE` once it has been modified. This is answered by a `TPrinter` object, an instance of which is returned by the `Printer` function in the `Printers` unit. It automatically gets a `DEVMODE` record using the appropriate API calls and maintains a reference to it via a handle

to the global memory block it occupies.

The TPrinter class defines two methods, GetPrinter and SetPrinter, which can be used to gain access to that DEVMODE, and update it after setting new field values. Ignore the help for these methods, which is currently incorrect. It suggests you should never need to call these methods, but the author of that text was mistaken.

Unusually for a VCL method, GetPrinter takes three PChar parameters, which are filled in with strings that describe the currently selected printer device, the printer driver (which is often blank) and the port used by the printer. Life is easiest if you use zero-based Char arrays for these PChar parameters, otherwise you will be obliged to allocate space for each of them using memory management routines. GetPrinter also takes a var parameter which is given the handle to the DEVMODE record.

Assuming a non-zero handle is returned, it can be turned into a pointer using the GlobalLock API, which must be paired with a call to GlobalUnlock when we are finished. When we have a valid pointer, this can be used to access the DEVMODE fields, as described earlier.

A simple project, which I have called PrinterSettings.dpr, is supplied on this month's companion disk, which shows how this code works. A button shows the current application printer settings using a printer setup dialog, and a radio group allows the paper orientation to be switched. When a radio button is selected, the current DEVMODE is accessed. Assuming orientation changes are supported, the orientation is toggled to the appropriate value and finally the modified DEVMODE is re-assigned to the printer. The code can be seen in Listing 7.

```
procedure TForm1.rgLocalOrientationClick(Sender: TObject);
var
  Device, Driver, Port: array[0..255] of Char;
  DevModeHdl: THandle;
  DevModePtr: PDevMode;
begin
  Printer.GetPrinter(Device, Driver, Port, DevModeHdl);
  if DevModeHdl <> 0 then begin
    DevModePtr := GlobalLock(DevModeHdl);
    if Assigned(DevModePtr) then
      try
        if DevModePtr^.dmFields and dm_Orientation = 0 then
          raise EPrinter.Create('Custom paper orientations not supported');
        case (Sender as TRadioGroup).ItemIndex of
          0: DevModePtr^.dmOrientation := DMORIENT_PORTRAIT;
          1: DevModePtr^.dmOrientation := DMORIENT_LANDSCAPE;
        end
      finally
        GlobalUnlock(DevModeHdl)
      end;
    Printer.SetPrinter(Device, Driver, Port, DevModeHdl);
  end
end;
```

➤ *Listing 7: Changing the paper orientation programmatically.*

The other thing to address was changing the global settings. Again, the usual way to do this is with an appropriate dialog so the user has control. To invoke the printer settings property sheet (which is what the dialog is referred to as) in a manner that allows updating the global settings, you can use the PrinterProperties API, declared in the WinSpool import unit. Another button in the sample project calls this using the code in Listing 8.

PrinterProperties requires a window handle that represents a form that the property sheet (or settings dialog) will be modal relative to, and also a handle to the printer whose settings you want to change. A TPrinter object has a handle property, but this is actually a device context handle (a handle to the printer's canvas) rather than a printer handle. TPrinter does maintain a printer handle internally, but it is not surfaced by any methods or properties, so we have to get one for ourselves using OpenPrinter.

```
procedure TForm1.btnGlobalSettingsClick(Sender: TObject);
var
  Device, Driver, Port: array[0..255] of Char;
  DevModeHdl, PrinterHandle: THandle;
begin
  //Get printer device name
  Printer.GetPrinter(Device, Driver, Port, DevModeHdl);
  //Get printer handle
  Win32Check(OpenPrinter(Device, PrinterHandle, nil));
  try
    //Invoke the printer property sheet
    Win32Check(PrinterProperties(Handle, PrinterHandle));
  finally
    //Close the opened handle
    ClosePrinter(PrinterHandle)
  end;
end;
```

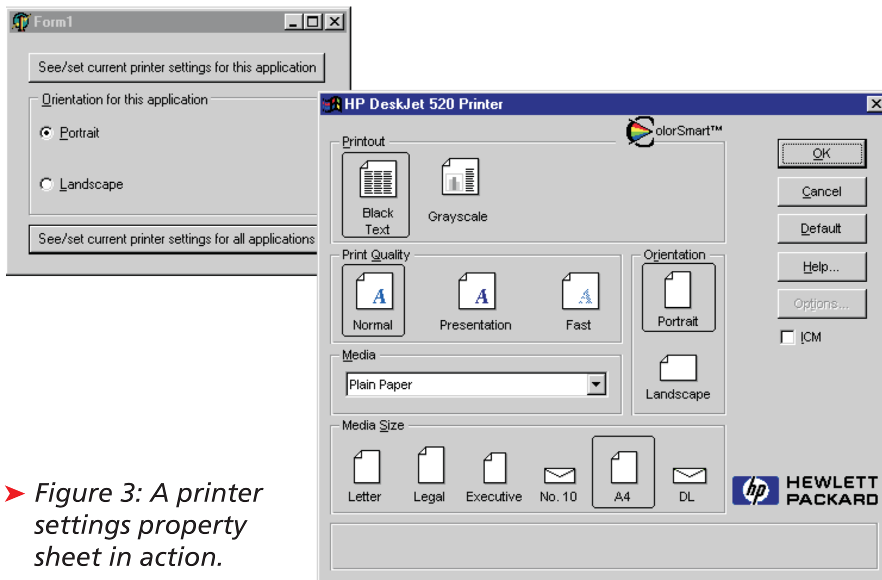
➤ *Listing 8: Changing the global printer settings via the printer property sheet.*

OpenPrinter requires a printer device name, which is not the same as the strings maintained in the TPrinter.Printers string list (which are readable combinations of both the device and the port), so we get the device from another call to GetPrinter. PrinterProperties does all that is necessary to change the global printer settings, so the only other thing we must do is close the printer handle that we opened, using ClosePrinter. Figure 3 shows an example of a printer's settings property sheet.

As for programmatically updating the global settings, there seems to be little information on this subject around, apart from some new support for this task in Windows 2000. I think the idea is that applications are encouraged not to take charge of changing these global settings: this should generally be left to the user.

However, I can see a case where it might be desirable. An application could be using some third-party DLL or application that performs some printing, and might wish to pre-set the printing options before it starts. Unfortunately, there seems to be no dedicated way to accommodate this (that I have found).

There is an API called DocumentProperties that allows modifications to be made to



➤ *Figure 3: A printer settings property sheet in action.*

printer settings, but it turns out to only modify application printer settings, not the global ones. Windows 2000 does add support for changing global settings, but that leaves Windows 95, 98 and NT without a solution.

One possible lead is that some printer drivers do store their data in dedicated areas of the Windows registry. For a given printer `XYZ` you might find its settings in:

```
HKEY_LOCAL_MACHINE\System\
   CurrentControlSet\control\
   Print\Printers\XYZ
```

In this registry key is a binary entry called `Default DevMode`, which should contain the printer's default settings in a `DEVMODE` record. So it would seem that you could read this value into an appropriately sized memory block, play with the appropriate fields, then write it back to the registry.

However, some drivers have no information stored in `Default DevMode`, preferring to use an INI file. An example is Adobe Acrobat's PDF Writer printer driver, which uses the PDFWRITR.INI file in the Windows System directory.

In short, I think you are out of luck if you are looking for a generic solution to change global printer settings on any Win32 platform.

Debugger Glitch

Q Something seems to be wrong with my Delphi debugger. Historically, as I went stepping through my code, a green arrow in the editor showed me which line was about to be executed. The editor no longer shows me this arrow and, as you can imagine, this makes code stepping quite difficult. How can I get the arrow back?

A The most likely cause of this problem is that the editor's gutter is too narrow (the gutter is the column down the left hand side of the editor). Since you mentioned the arrow used to be displayed as green, this implies you are using Delphi 3 or later.

You can change the gutter's width with the `Gutter width:` option on the `Display` page of the environment options dialog in Delphi 3 and 4, or the editor options dialog in Delphi 5 and later. The default value is 30 pixels.

IDE Anomaly

Q I use the `Minimize On Run` option to keep my Windows environment nice and tidy whilst testing my applications. Recently, though, Delphi has stopped restoring itself when the program is closed and I have to click on the Delphi task bar icon myself. I cannot work out what I have changed, so can you help me?

A The `Minimize On Run` option in the environment options dialog is fairly simplistic in its behaviour. When the application is successfully launched, the IDE minimises itself. In order to restore itself when the program closes, it must somehow be notified when this happens. The way it does this is by using the integrated debugger.

The debugger is notified when many 'interesting' things happen to the program, and its termination is just one of them. When the debugger notices the program has terminated, the IDE is able to restore itself. If the IDE is not restoring itself successfully, the implication is that you have turned off integrated debugging. Turn it back on and all will be fine.

Debugger Failure

Q I have a problem trying to get Delphi programs to compile and run with the integrated debugger switched on. The following message appears when I try and run the program: *Debugger Kernel BORDBK50.DLL is missing or not registered*.

The referenced DLL is on my hard disk in C:\Program Files\Common Files\Borland Shared\Debugger, and was placed there during installation. Maybe you could shed some light on the problem (I have a feeling it is a problem with Windows 2000, which is giving me trouble in other areas as well).

A Fortunately, this is not a problem specific to Windows 2000. In fact I have seen this error a few times, on various Win32 platforms (Figure 4).

BORDBK50.DLL is an in-process COM server which implements version 5 of Borland's Win32 integrated debugger. For some reason, the installation fails to successfully register this server sometimes, meaning you have to do it yourself.

Thankfully, this is an easy task. You can either use Borland's TRegSvr.exe utility (supplied in Delphi's Bin directory) or the Windows utility RegSvr32.exe. From a command prompt, navigate your way to the directory

locating the DLL. Then pass the DLL name as a command-line parameter to either of the aforementioned registration utilities, for example:

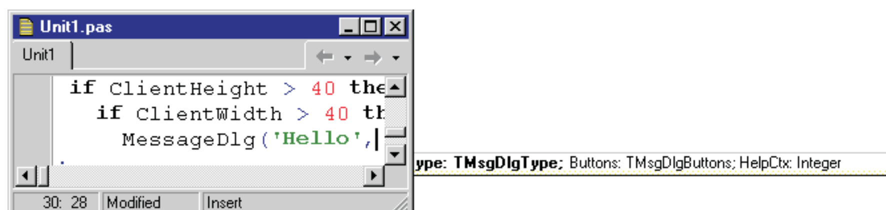
```
RegSvr32 BorDbk50.dll
```

Missing Code Parameters

Q I have been using Delphi for some while and have noticed an intermittent problem with the Code Parameters tooltip that shows subroutine parameters in the editor. I know that it doesn't appear when there are errors in the preceding code, but occasionally it seems to stop working even when I know that it should work. I've checked the option on the `Code Insight` page of the editor options dialog and that is still enabled, so how do I fix it?

A I've had this problem myself whilst running Delphi on Windows 95 and have seen it on Windows 98, but I am not sure if it also occurs on Windows NT/2000. As the question suggests, the Code Parameters window seems to stubbornly refuse to appear when you know it should do.

➤ *Figure 4: The debugger failing to start.*



➤ *Figure 5: The Code Parameters window misbehaving.*

In fact, this diagnosis is incorrect, as I found out one day by chance when I had shrunk my code editor to a very small size. In fact, the Code Parameters window does appear, but it loses its stay-on-top attribute and ends up stuck behind the editor (see Figure 5). Whilst I still don't know why the problem occurs, I have managed to come up with a couple of workarounds for it.

The easiest one is to close Delphi and restart it. But if that seems undesirable for any reason, I have an alternative quick fix. The unit FixCodeParams.pas on the disk (see Listing 9) can be added to a new package in the IDE and installed by pressing the package editor's `Install` button. Once installed, the problem will be solved by the code in the `Register` routine so it can then be uninstalled via the `Component | Install Packages...` dialog.

The code searches for the Code Parameters window (which is owned by the `Application` object). If it finds it, the code proceeds to mark the window as a normal window (rather than a stay-on-top window), and then resets it to be a stay-on-top window. This seems to do the trick for my system: simple. The package can be kept around in case the problem resurfaces at a later date.

➤ *Listing 9: Retrieving a missing Code Parameters window.*

```
unit FixCodeParams;
//This unit fixes the problem of the Code Parameters window not appearing
// 1) Add this unit to a new package
//    (File | New... | Package)
// 2) Install the package (press the package editor's
//    Install button). The problem will now be fixed.
// 3) Uninstall the package
//    (Component | Install Packages..., select the package
//    from the list and press Remove)
interface
procedure Register;
implementation
uses
  Windows, Controls, SysUtils, Forms;
procedure Register;
var
  I: Integer;
  R: TRect;
  CodeParams: TCustomControl;
begin
  for I := 0 to Application.ComponentCount - 1 do begin
    if (CompareText(Application.Components[I].Name, 'CodeParamWindow') = 0) and
       (CompareText(Application.Components[I].ClassName, 'TTokenWindow') = 0)
      then begin
      CodeParams := Application.Components[I] as TCustomControl;
      GetWindowRect(CodeParams.Handle, R);
      SetWindowPos(CodeParams.Handle, HWND_TOP, R.Left, R.Top, R.Right-R.Left,
        R.Bottom-R.Top, SWP_NOACTIVATE);
      SetWindowPos(CodeParams.Handle, HWND_TOPMOST, R.Left, R.Top,
        R.Right-R.Left, R.Bottom-R.Top, SWP_NOACTIVATE);
      Break
    end
  end;
end;
end.
```