Beating the System: Delphi Meets IntelliMouse

Adding IntelliMouse support to your Delphi applications

by Dave Jewell

Having recently taken delivery of a new Dell PC *[was that a gloat or three I heard there, Dave? Ed]*, and spent some time playing with the Microsoft IntelliMouse which accompanied it, I decided that it would be fun to write an article on how to add IntelliMouse support into your own Delphi applications.

As you will doubtless be aware, the latest Microsoft mouse is unusual in that it includes a small rubberised wheel located between the two conventional mouse buttons. If you're using a recent Microsoft package, such as Office 97, you'll know that this wheel can be used to scroll the active window in the current program. This is very convenient because, under normal circumstances, if you're working with a maximised application and you wish to scroll the window, you have to move the mouse well away from the area where you're working in order to reach the scroll bar at the extreme right-hand edge of the screen. With the clever new mouse wheel, you can scroll the window without moving anything except your index finger. This might sound like the ultimate in slothfulness, but once you've tried it you won't want to go back to the old way!

And Then There Were Three... Again!

So how easy is it to add IntelliMouse support to a Delphi application? In this article, I'm going to provide code which will enable you to do just that. Let's get the really simple stuff out of the way first. Although it's not obvious until you've pressed it, the IntelliMouse is actually a three button mouse: the third (middle) button is located under the wheel itself and you press the button by depressing the wheel rather than rolling it.

It's amusing to note that, in this respect, Windows has gradually come full circle. Back in the early days of Windows, three-button mice were relatively commonplace. The more buttons a mouse had, the better it was (at least, according to the three-button mouse manufacturers). But the public were not fooled: they soon realised that very little software was written to take advantage of the third button. In fact, very little software was written to use the *second* button on a mouse! Indeed, if you're as long in the tooth as I am, you may remember that under Windows 1.0, Microsoft recommended that software should be designed to work without any mouse at all. The mouse was very much an optional extra. It wasn't until the relatively recent launch of Windows 95 that Microsoft decided to make serious use of the second button through (for example) context menu handlers in the shell and popup menus in their Office suite.

Responding to the third mouse button is the easiest aspect of IntelliMouse compatibility. That's because the messages generated by the third mouse button are simply the old, familiar third-button messages which have been built into Windows since version 1.0. These are defined in the file MESSAGES.PAS file, as shown in Listing 1.

Because Delphi already understands these messages, we can make use of them by just writing a message handler procedure in the usual way. Suppose, for example, that you're writing a drawing application and you want to use the middle mouse button to select all the objects on the current page. Listing 2 shows how easily this could be done.

In the same way, you can do custom things when the middle button is released and when it's double-clicked.

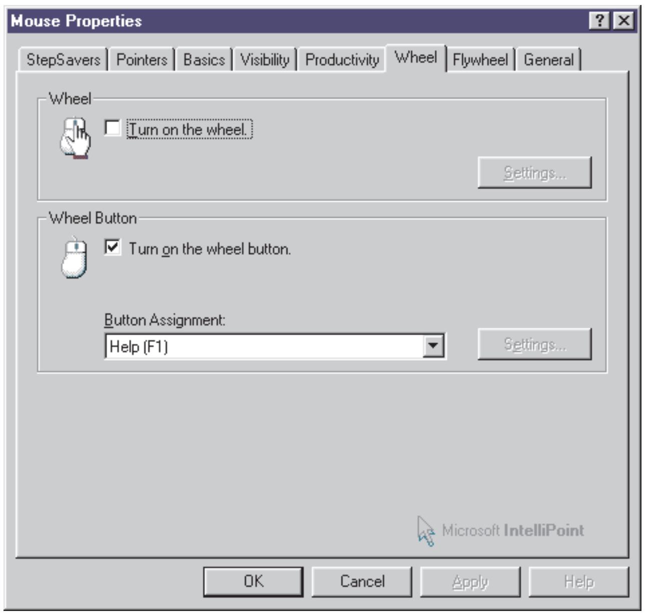
There are a couple of important points to make here. Firstly, you should be clear that these middle-button messages will be generated for any three-button mouse. This is why I say that supporting the third button is the easiest aspect of IntelliMouse support: you don't need to check for the presence of an IntelliMouse in the way that (as you'll see in a moment) we need to

➤ *Listing 1*

```
WM_MBUTTONDOWN   = $0207;  // middle button down
WM_MBUTTONUP     = $0208;  // middle button up
WM_MBUTTONDBLCLK = $0209;  // middle button double-click
```

➤ *Listing 2*

```
TsuperDraw = class (TForm)
private
  { Private declarations }
  procedure MiddleButtonClicked (var Msg: TMessage); message wm_MButtonDown;
end;
...
procedure MiddleButtonClicked (var Msg: TMessage);
begin
  // Code to select all objects goes here...
end;
```



Wheel

☐ Turn on the wheel.

Wheel Button

☑ Turn on the wheel button.

Button Assignment: Help (F1)

➤ *Figure 1: This is the Mouse Properties dialog accessed from the Mouse applet in the Control Panel. If Microsoft's IntelliPoint software is in use you can change a number of IntelliMouse options.*

do when using the mouse wheel. Your program doesn't know, and shouldn't care, whether those middle button messages are generated by an IntelliMouse or by an ancient three-button mouse that was produced when Windows 1.0 was young. Thus, you *cannot* assume you've got an IntelliMouse just because you're receiving middle button messages.

Secondly, you should be aware that the user may have configured the middle button to do something non-standard. For example, look at Figure 1. This shows the Mouse Properties dialog which is displayed when you select the Mouse applet in the Control Panel. Personally, I like to enable the middle button and specify that I want it to return an `F1` keystroke. You'll find that this is terrific when working in the Delphi IDE: you can double-click the left mouse button to highlight a function name and then press the mouse wheel to bring up context-sensitive help on that function, all with no wrist-action whatsoever!

So Let's Get Rolling....

Of course, the real fun starts when we begin interacting with the mouse wheel. In order to do this, we need to determine whether an IntelliMouse is installed and whether wheel support is currently enabled. The bad news is that Windows 95 doesn't support the IntelliMouse *natively* (to use Microsoft's jargon). What they mean is that mouse wheel support isn't built-in and doesn't happen automatically. Currently, Windows 95 doesn't provide built-in support for the IntelliMouse, but NT 4.0 does. As you'd expect, both NT 5.0 and Windows 98 will include built-in support. Does this mean we can't use the mouse-wheel under Windows 95 or pre-4.0 versions of NT? No, it just means we have a little extra work to do.

Take a look at Listing 3, which illustrates the basic idea. This is a complete, ready-to-run program called MouseDemo, the source code for which is included on this month's disk. This code illustrates how to receive mouse wheel messages in an operating system independent manner.

As you can see, the `FormCreate` routine calls another method, called `IntelliMouseInit`, which is where the real work gets done. Inside `IntelliMouseInit`, the first thing we do is invoke the local `NativeMouseWheelSupport` routine to determine if we're running an operating system that provides native mouse support. This code simply uses the API `GetVersionEx` call to determine the operating system version: if it detects that we're running on NT version 4.0 or later, it gives the thumbs up. Similarly, the green light is given if it detects that we're running Windows version 5.0 (also known as Windows 98) or higher.

Note that the code I've written assumes that the shrink-wrap version of Windows 98 is going to return a major version ID of 5. This is consistent with Windows 95 returning 4. Unfortunately, the current version of Windows 98 (beta 3) resolutely continues to return a major version ID of 4, and you'll see that I've deliberately added a most unpleasant hack to get around this. At some future point, you should modify the `NativeMouseWheelSupport` routine according to whatever final version numbers Microsoft implement for Windows 98.

If the `NativeMouseWheelSupport` function returns `True`, then things are dead easy: we can call the `GetSystemMetrics` routine, passing it a value of `sm_MouseWheelPresent` to determine if an IntelliMouse is installed. Similarly, calling `SystemParametersInfo` with the `spi_GetWheelScrollLines` specifier will tell us how many lines to scroll a window for each 'nudge' of the mouse wheel (this is user-configurable in the Mouse Properties dialog via the Control Panel applet). Finally, if native mouse wheel support is provided by the operating system, we know that when the wheel is turned, the active window will receive Windows messages with the value `wm_MouseWheel`.

Easy peasy, eh? Unfortunately, things are a little more complex when the operating system doesn't provide native IntelliMouse support. Under these circumstances, we have to do a little spade-work for ourselves. The first job is to search for a hidden top-level window with the name `MouseZ` and the class `Magellan MSWHEEL`. This window is created by Microsoft's bolt-on (ie non-native) IntelliMouse support software. If the window doesn't exist, then we can forget the whole deal, mouse wheel support isn't installed. If the window is found, then we need to send a couple of messages to the window in order to get the information we need.

To do this, we need to make use of the `RegisterWindowMessage` routine, an API call which you may not be familiar with, since it's not often used. In a nutshell, you call this routine passing it the name of a custom message that you want to define. In turn, the routine will give you back an integer value which you should use in subsequent SendMessage calls (assuming you generate the message) or should look out for in your window procedure (assuming you receive the message). Microsoft adopted this strategy because they obviously didn't want people defining their own custom Windows message numbers: all sorts of conflicts would arise. In order for this scheme to work properly, RegisterWindowMessage must return the same message number for two different processes which register the same message name. In this way, communication can then be established between different processes. All they have to agree on is a message name: no hard-wired message number is required.

To determine if wheel support is enabled, we have to send a special message to the hidden IntelliMouse window. But the number of this message isn't defined by Windows itself: we can only discover it by calling RegisterWindowMessage with the pre-defined MSH_WHEELSUPPORT_MSG string. Having got the corresponding message number, we then send this message to the hidden window, checking the result of the SendMessage call. If true, then mouse wheel support is enabled.

But we're not quite out of the woods yet. We also have to query the invisible window to determine the number of scroll lines which correspond to each wheel 'nudge' and to do that (you've guessed!) we need to send another special message number, which necessitates another call to the RegisterWindowMessage routine. Finally, we call this same API routine one more time in order to get the custom message number used by the IntelliMouse software when sending mouse wheel movement notifications to the focused window: this corresponds directly to wm_MouseWheel message on platforms that natively support the IntelliMouse.

At this point, you're probably asking yourself why Microsoft did things this way? Why couldn't they make things look the same even on the non-native platforms? Well, I

➤ *Listing 3*

```
unit UMouseDemo;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
  Panel1: TPanel;
  procedure FormCreate(Sender: TObject);
  private
    // True if IntelliMouse + wheel enabled
    fIntelliWheelSupport: Boolean;
    // message sent from mouse on wheel roll
    fIntelliMessage: UINT;
    // number of lines to scroll per wheel roll
    fIntelliScrollLines: Integer;
    procedure IntelliMouseInit;
    procedure WndProc(var Message: TMessage); override;
    procedure WMMouseWheel(var Message: TMessage);
      message wm_MouseWheel;
  public
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  IntelliMouseInit;
end;
procedure TForm1.WndProc(var Message: TMessage);
  function GetShiftState: Integer;
  begin
    Result := 0;
    if GetAsyncKeyState (vk_Shift)   < 0 then
      Result := Result or mk_Shift;
    if GetAsyncKeyState (vk_Control) < 0 then
      Result := Result or mk_Control;
    if GetAsyncKeyState (vk_LButton) < 0 then
      Result := Result or mk_LButton;
    if GetAsyncKeyState (vk_RButton) < 0 then
      Result := Result or mk_RButton;
    if GetAsyncKeyState (vk_MButton) < 0 then
      Result := Result or mk_MButton;
  end;
begin
  { If message is non-native, eat non-native message and post
    a native message. We don't call Inherited, thus ensuring
    original message is discarded. }
  if (Message.Msg = fIntelliMessage) and
    (fIntelliMessage <> wm_MouseWheel) then begin
    { Need to convert non-native info into native format }
    PostMessage(Handle, wm_MouseWheel, MakeLong(
      GetShiftState, Message.wParam), Message.lParam);
  end else
    Inherited;
end;
procedure TForm1.WMMouseWheel(var Message: TMessage);
begin
  // Application specific code goes here...
end;
procedure TForm1.IntelliMouseInit;
var
  hWndMouse: hWnd;
  mQuerySupport: UINT;
  mQueryScrollLines: UINT;
  function NativeMouseWheelSupport: Boolean;
  var
    ver: TOSVersionInfo;
  begin
    Result := False;
    ver.dwOSVersionInfoSize := sizeof (ver);
    // For Windows 98, assume dwMajorVersion = 5
    // For NT, we need 4.0 or better.
    if GetVersionEx (ver) then case ver.dwPlatformID of
      ver_Platform_Win32_Windows :
        Result := ver.dwMajorVersion >= 5;
      ver_Platform_Win32_NT      :
        Result := ver.dwMajorVersion >= 4;
    end;
    { Quick and dirty temporary hack for Windows 98 beta 3 }
    if (Result = False) and
      (ver.szCSDVersion = ' Beta 3') then
      Result := True;
  end;
begin
  if NativeMouseWheelSupport then begin
    fIntelliWheelSupport :=
      Boolean (GetSystemMetrics (sm_MouseWheelPresent));
    SystemParametersInfo(spi_GetWheelScrollLines, 0,
      @fIntelliScrollLines, 0);
    fIntelliMessage := wm_MouseWheel;
  end else begin
    { Look for hidden mouse window }
    hWndMouse := FindWindow ('MouseZ', 'Magellan MSWHEEL');
    if hWndMouse <> 0 then begin
      { OK, window is there but is wheel support enabled? }
      mQuerySupport :=
        RegisterWindowMessage('MSH_WHEELSUPPORT_MSG');
      if Boolean(SendMessage(hWndMouse, mQuerySupport,
        0, 0)) then begin
        { We're in business get the scroll line info }
        fIntelliWheelSupport := True;
        mQueryScrollLines :=
          RegisterWindowMessage ('MSH_SCROLL_LINES_MSG');
        fIntelliScrollLines :=
          SendMessage (hWndMouse, mQueryScrollLines, 0, 0);
        { Finally, get the custom mouse message as well }
        fIntelliMessage :=
          RegisterWindowMessage ('MSWHEEL_ROLLMSG');
      end;
    end;
  end;
  if (fIntelliScrollLines < 0) or
    (fIntelliScrollLines > 100) then
    fIntelliScrollLines := 3;
end;
end.
```

accept that patching the GetSystemMetrics and SystemParametersInfo routines *in situ* would have been messy, but why the dickens didn't they specify that the mouse wheel notification message would be $020A on both native and non-native platforms? As I've observed in previous articles, Microsoft are not exactly in the first rank when it comes to making life easier for the developer...

OK, so we've called IntelliMouseInit and we know that our form is going to receive special mouse wheel notification messages, the message number being stored in the fIntelliMessage field. Unfortunately, Object Pascal won't let us do this:

```
procedure WMMouseWheel(var
  Message: TMessage);
  message fIntelliMessage;
```

In other words, we can't use the contents of a variable as the message index, we have to come up with something that the compiler can evaluate as a constant (this is why I say life would have been simpler if Microsoft had used wm_MouseWheel for both native and non-native cases). In order to get around this problem, I decided to override the form's WndProc routine as shown in Listing 3. The WndProc method effectively gets first crack at all the messages sent to a particular VCL windowed control. Here, I check if the message being received is the custom mousewheel notification. If it is, and if we're running non-native (the message isn't equal to wm_MouseWheel), then I throw away the message and post a new message using the native message format. See later for an explanation of what I mean by this. In effect, the WndProc code ensures that the application gets sent the same message number, in the same format, irrespectively of whether or not it's running on a platform that provides native support.

And now we're home and dry. With this change, you can write a message handler, WMMouseWheel as shown in the code listing, which responds to wm_MouseWheel messages. This handler doesn't know or care whether the messages it receives were generated natively or posted from the WndProc code, the remainder of the application code can be platform independent.

Anatomy Of The wm_MouseWheel Message

So what exactly is contained in the wm_MouseWheel message? Firstly, the mouse co-ordinates are stored in the low and high words of the lParam field, the horizontal co-ordinate being in the low-order word and the vertical co-ordinate in the high-order word. As with all wm_MouseXXX messages, these are screen co-ordinates, ie relative to the top-left corner of the screen.

In addition, this message also encodes a 'notch' count, and an indication of what modifier keys were pressed when the mouse wheel rotation took place. The Microsoft IntelliMouse has 18 little 'notches' (the fancy American word is *detents*) spaced evenly around the mouse wheel. This, of course, means that each notch corresponds to a 20 degree rotation of the wheel. The mouse is designed in such a way that you can't get any finer resolution than this: if you stop the wheel between notches, nothing will happen. Although I haven't yet stripped my mouse down to its bare essentials, I imagine this is achieved using a small micro-switch making contact with an 18-toothed cog.

The wm_MouseWheel message signifies not only that the mouse wheel has been rotated, but it also tells the application how many notches have been traversed. In other words, you shouldn't assume that you'll get one message for each notch. The Microsoft documentation is vague but the implication is that if the wheel is turning quickly, the operating system might bundle up a number of 'notches' into a single wm_MouseWheel message. So how does this message encode the number of notches? Thereby hangs a tail! (Get it? Mouse? Tail? Oh, please yourselves...) *[Never knew you were a Frankie Howerd fan, Dave! Ed].*

The notch count is always multiplied by a factor of 120. Thus, if the wheel is moved by one notch, then we'll receive a notch count of 120, two notches and we'll get a value of 240, and so on. What's the reason for this? Well, in this particular case, there is some method to Microsoft's madness! When designing the IntelliMouse API, they chose to allow for the possibility of higher resolution mouse wheels being produced in the future, either by Microsoft or by others. I've already mentioned that a single notch corresponds to a 20 degree rotation of the mouse wheel. Now imagine some future mouse wheel that gives a resolution of two degrees or 180 notches. In this case, the mouse wheel might generate a wm_MouseWheel message which returns a value of 12 (120/10 = 12) for each notch that's traversed. If you want to look at things more simply, just think of the IntelliMouse as reporting wheel rotation in units of one sixth of a degree.

The general idea is that the value of 120 represents a threshold below which you should ignore the message. If you want to be absolutely correct, your software shouldn't assume that it will receive rotation counts of at least 120. Instead, it should accumulate rotation counts until at least a value of 120 is reached. At this

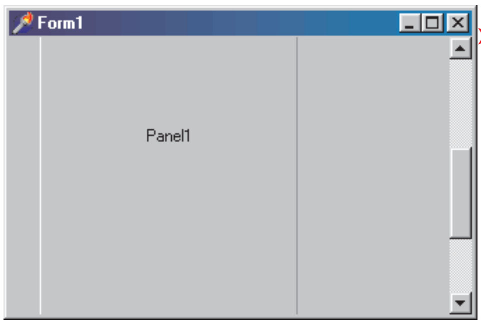


Figure 2: Just an over-sized TPanel being used to enable the vertical scrollbar of a TForm, but it shows how to implement IntelliMouse scrolling in your Delphi programs under platforms that do and don't support the mouse wheel natively.

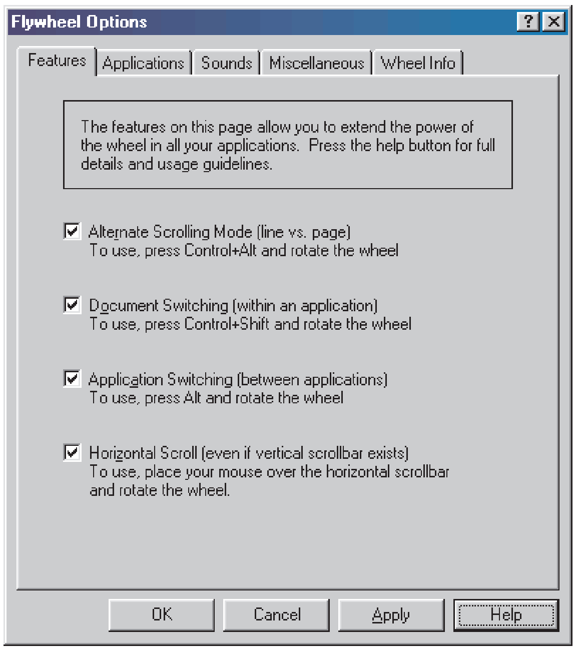
point, it should take some application-specific action (usually scrolling a window) and saving any residual rotation count that exceeded the threshold of 120. In this way, your software will perform identically whether using low or high-resolution mouse wheels. Another approach (assuming that you are scrolling a window) is to make use of sub-120 values to perform partial line scrolls.

Thus, if you get a value of 12, you would scroll by one tenth of a line height, 60 would scroll by half a line height and so forth. This will give a very smooth effect when working with future high-resolution mice. Of course, some software (notably graphics drawing packages) will want to make full use of high-resolution mice, but that's an application-specific issue.

In another brilliant demonstration of programming acumen, Microsoft have encoded the notch information *differently* for operating systems that either do or don't support the IntelliMouse natively. You'd think that after the `RegisterWindowMessage` obstacle course, the folks from Redmond would have relented and made things nice and straightforward? No, not a bit of it... Read on... *[Ah, but think how bored you'd get without all these little challenges! Ed]*

Under non-native platforms (Windows 95 and pre-4.0 versions of NT) the notch count information is contained in the entire 32-bit `wParam` field of the message. However, under natively supporting platforms (Windows 98, NT 4.0 and later) the notch count is contained

➤ *Figure 3: Some shareware utilities install a global message hook which enables even non-IntelliMouse aware applications to work with the mouse wheel. FlyWheel 2 from Plannet Crafters is one such example. You can download an evaluation version of FlyWheel 2 from the Plannet Crafters website at www.plannetarium.com*



The features on this page allow you to extend the power of the wheel in all your applications. Press the help button for full details and usage guidelines.

☑ Alternate Scrolling Mode (line vs. page)
To use, press Control+Alt and rotate the wheel

☑ Document Switching (within an application)
To use, press Control+Shift and rotate the wheel

☑ Application Switching (between applications)
To use, press Alt and rotate the wheel

☑ Horizontal Scroll (even if vertical scrollbar exists)
To use, place your mouse over the horizontal scrollbar and rotate the wheel.

in the high-order 16 bits of the `wParam` field, with the modifier keys (control, shift, etc) in the low-order word. For non-native platforms, you have to figure out the modifiers for yourself. In fairness, it's perhaps best to assume that this is a genuine mistake on Microsoft's part and that some sort of error was made in the Windows NT code. I find it very hard to believe that they really intended it should work this way.

The final piece in the jigsaw, in case you were wondering, is the direction of wheel rotation. This is very simple: a positive value for the notch count indicates that the top of the wheel is moving away from the user while a negative value implies that the wheel is moving towards the user.

Putting this all together, a typical `WMMouseWheel` handler might look something like that shown in Listing 4. This code makes use of a typed constant (a persistent variable which will hold its value from one invocation of the method to the next) to store the accumulating delta value. This caters for the use of possible future high-precision mouse wheels. When the value of `Delta` exceeds 120, then the remaining code is triggered. Notice the presence of the `while` loop: this likewise caters for the possibility that a large `Delta` value has been delivered by a single message as when (allegedly) the mouse wheel is moving at speed. You'll also notice the two inner `for` loops which generate as many `wm_VScroll` messages as are implied by the value of the scroll-lines variable.

This implementation is only provided as an example: how you implement the `WMMouseWheel` handler will depend very much on your own individual application. When you start developing under a platform that provides native mouse wheel support, you will find that a number of the standard Windows controls already have built-in IntelliMouse support. The list-box control is a good example of this: you can take an existing Delphi application which knows nothing about the IntelliMouse, and you'll find that under Windows 98 (for example) the mouse wheel will scroll a list-box, provided that the list-box has the input focus.

➤ *Listing 4*

```
procedure TForm1.WMMouseWheel(var Message: TMessage);
const
  Delta: SmallInt = 0;
var
  Idx: Integer;
begin
  Delta := Delta + HiWord (Message.wParam);
  while Abs(Delta) >= 120 do begin
    if Delta < 0 then begin
      for Idx := 0 to fIntelliScrollLines - 1 do
        PostMessage (Handle, wm_VScroll, MakeLong (sb_LineDown, 0), 0);
      Delta := Delta + 120;
    end else begin
      for Idx := 0 to fIntelliScrollLines - 1 do
        PostMessage (Handle, wm_VScroll, MakeLong (sb_LineUp, 0), 0);
      Delta := Delta - 120;
    end;
  end;
end;
```

This works because the VCL `TListBox` implementation is only an object-oriented wrapper around the API-level list-box code within USER.EXE. Microsoft have modified the window procedure for this control so as to recognise and act upon `wm_MouseWheel` messages, hence instant support for the IntelliMouse. Of course, this only works for the VCL classes that directly wrap the intrinsic Windows controls. A `TScrollBox` will studiously ignore the mouse wheel, as will a form which is currently displaying scroll bars.

In next month's *Beating the System* I'll finish off this IntelliMouse tutorial by packaging up the existing code into a re-usable component, making it much easier to incorporate into an existing program, and I'll also be describing how to add IntelliMouse support to the Delphi 3.0 IDE through the use of a special package.

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