How to…
Write applications using Visual Basic

Last month, I finished explaining some of the properties, methods and events that most VB controls have in common, and we started to write our "Doodler" program to put what you've learnt into practice. Continuing from last month, you should have already created the *user interface* (the form and its controls) and now we're ready to start adding the logic behind them.

Keep it colourful

We'll begin by adding the names of some colours into the *lstColours* listbox. The *ListBox* control shares a great deal in common with the *ComboBox*, which we've met earlier in this series. Just as we've used the *AddItem* method of the combo box to add choices to its pop-down list, we'll use the *AddItem* of the list box to add some colours to its list. Don't worry about the unfamiliar terms, I'll explain them in a moment. Enter the following code into the *Form_Load* event of *frmDoodler*:

```
With lstColours
    .AddItem "Black"
    .ItemData(.NewIndex) = RGB(0, 0, 0)
    .AddItem "Red"
    .ItemData(.NewIndex) = RGB(255, 0, 0)
    .AddItem "Green"
    .ItemData(.NewIndex) = RGB(0, 255, 0)
    .AddItem "Blue"
    .ItemData(.NewIndex) = RGB(0, 0, 255)
    .AddItem "Yellow"
    .ItemData(.NewIndex) = RGB(255, 255, 0)
    .AddItem "Cyan"
    .ItemData(.NewIndex) = RGB(0, 255, 255)
    .AddItem "Magenta"
    .ItemData(.NewIndex) = RGB(255, 0, 255)
    .AddItem "White"
    .ItemData(.NewIndex) = RGB(255, 255, 255)

    .ListIndex = 0
End With

optShape(0).Value = True
```

The last time we used the *AddItem* method, we looped around the elements of an array adding each item in turn. However, we're adding the items individually in this case since we're not using an array. Notice the use of the *With* keyword – this provides a convenient way to refer to the properties and methods of a control without having to keep

specifying its name. Once VB has encountered a *With* keyword, it assumes that any references to methods and properties that aren't prefixed with the name of a control are referring to the control that you mentioned in the preceding *With* statement. Once you've finished using this feature, you need to disable it using a corresponding *End With* as I have done. It would be just as valid to enter

```
lstColours.AddItem "Black"
lstColours.ItemData(.NewIndex) = RGB(0, 0, 0)
lstColours.AddItem "Red"
lstColours.ItemData(.NewIndex) = RGB(255, 0, 0)
...
..  (and so on)
```

omitting the *With* block altogether. However, I recommend that you use *With* wherever possible; not only is it less typing, but it's also marginally faster when VB gets round to executing your program.

Doing colours the Windows Way

If you have programmed using a Commodore 64 or Sinclair Spectrum before, you might be used to having a fixed colour palette, for example, colour 0 is black, colour 1 is white, colour 2 is red and so on. No such limitation exists on the PC; you are free to "invent" your own colours as you see fit. All you need to do is to state the intensity of red, green, and blue that you want in the colour, and it is made for you.

You refer to how much red, green and blue should be used by specifying a number from 0 to 255, where 0 is the least intensity of that colour and 255 is the greatest intensity of that colour. For example, a colour containing a red value of 255, a green value of 0 and a blue value of 255 would produce magenta since red mixed with blue makes magenta. Setting all the values to 0 produces black and setting them all to 255 produces white.

How do we use this in Visual Basic? Well, colours in VB are represented using a single 32-bit number rather than the three separate values for the red, green and blue components of a colour. Visual Basic provides the *RGB* function, which returns the appropriate 32-bit value when given the various intensities of Red, Green and Blue respectively. Using the example of magenta used earlier, we could specify RGB(255,0,255) which would return a value of 16711935, which to VB, means magenta.

You can also use *constants* rather than the RGB function to specify colour values if you want, although the RGB function is more flexible since you can specify colours that have no constant equivalent. See the "Constants" box elsewhere in this article for an explanation as to what a constant is.

One point whilst we're on the subject of colour – the colours on your screen may not exactly match what you ask for. This is dependent on the display driver and colour depth you use. To accurately reproduce the entire spectrum of colours, configure your display

for 16-bit (65536 colours), or preferably, 24-bit colour (16777216 colours) via the *Settings* tab of the *Display* applet in the Control Panel.

Now we know how to specify colours, we just need a way to associate them with their names in the ListBox, which brings me to the next subject…

Hidden Agenda

A new property has crept into the *Form_Load* code – *ItemData*. Every item in a ListBox (or ComboBox, for that matter) can have a hidden number associated with it – this number is accessed by the *ItemData* property. As luck would have it, this number is a 32-bit integral value, which makes it ideal for storing the *RGB* value of each colour. Therefore, when we add the description of the colour using the *AddItem* method, we immediately follow this by setting the *ItemData* property of the list item we've just added to reflect the RGB value of the colour it represents. Notice the use of the *NewIndex* property to get the list index of the item that was most recently added to the ListBox.

Finally, we set which items of the user interface will be initially selected when the program is started. We make the first item in the ListBox the default selection by setting the listbox's *ListIndex* property to 0 (don't forget that list items are numbered from 0, not 1). Then, we select the first *OptionButton* in the control array *optShape* to be the initially selected control within the array by setting its *Value* property to *True*. The *Value* property of an *OptionButton* returns *True* if it is selected or *False* if it is not. Please note that the *Value* property is the default property of an *OptionButton* so it could have been omitted. I've left it in for clarity in this case although you can remove it if you wish.

Drawing on a Picture(Box)

All our drawing will be taking place in the *PictureBox* we've called *picDrawingArea*. Why a *PictureBox*? Well, the *PictureBox* control is the only control that lets you draw onto it directly. We could draw directly onto our form, *frmDoodler* instead but then, we'd have to write some code to stop the user from drawing onto areas that we didn't intend, for example underneath the list of colours. Since the *PictureBox* provides a visible boundary to show the user whereabouts the drawing can take place, and it also supports being drawn onto directly, it is the perfect choice for our task.

Forgetful VB?

There's one more catch to drawing directly onto a control – by default, VB doesn't remember what has been drawn onto a control. If anything causes VB to redraw the PictureBox, for example placing another window over the top of it and then moving that window out of the way again, our work of art will be erased. This isn't VB's fault – it's

the way that Windows works, and many other graphical operating systems along with it for that matter. Behind the scenes, when one window overlaps another, the bit that is covered up isn't remembered. When the user moves the topmost window elsewhere and exposes the region of the underlying window was previously covered, Windows asks the underlying window to redraw its newly "exposed" region. Since we're not going to be storing a separate copy of what the user draws, we won't be able to do this because we won't know what was there previously. Fortunately, VB provides the *AutoRedraw* property – this tells VB to keep a copy of whatever we draw onto a control. If we set this property on our *PictureBox* to *True*, VB will remember what was there previously and do all the necessary redrawing for us.

In Closing

That's all for this month – as usual, you can find the project files that accompany this tutorial on the cover disc. Next month, we'll continue with the Doodler program and add the code that performs the actual drawing itself.

Good luck,
Nick.

Nicholas Scott is a freelance columnist who currently works for MIS Computer Services in Northwich. Nick can be contacted via email at nicks@miscs.com.

Those missing bits…

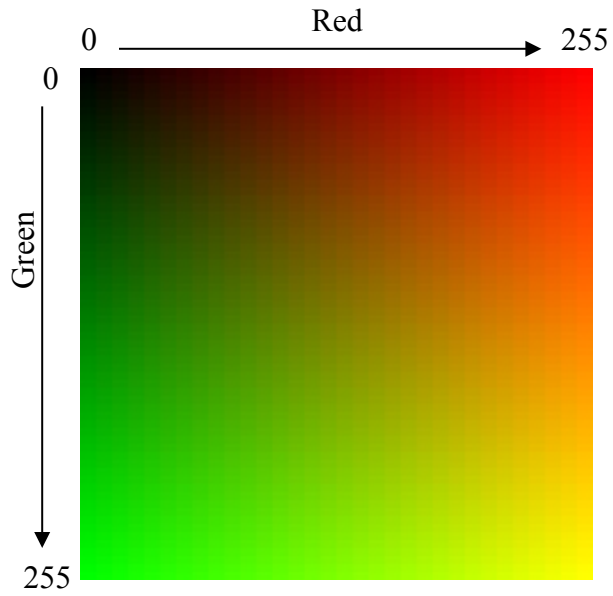
The techies amongst you might wonder why I said that colours are represented by a 32-bit value. After all, colours are represented by three 8-bit values and 8+8+8 makes 24 bits, not 32. The answer is, that Windows itself uses the other 8 bits of the value to assist it in representing the so-called "System Colours". System colours are used to draw things which have no fixed colour such as the window frames, the desktop background colour and so on. I say no fixed colour, because you can define which colours are used for drawing these things via the *Appearance* tab in the *Display* applet of the Control Panel. If you refer to the system colours in your program, you'll get the appropriate colours from the control panel instead. To see a list of these system colours, click a VB form, click its *BackColor* property and then click the combo box that appears. Choose a colour and then look at the hexadecimal value that VB puts into the *BackColor* property. If you wanted to use a system colour, you'd have to specify this hexadecimal equivalent directly since the RGB function only returns 24-bit values. Or better still, use a constant – see the box on constants elsewhere in this article for details.

Constants

Constants are like variables, except that their values can't be changed, i.e. they're constant. Constants are useful because you can refer to certain "magic" values using constants instead, making your programs more readable. For example, the value 16711935 means "magenta" to VB when used as the value for a colour. However, Visual Basic already has a constant for this called *vbMagenta*, so you could use this instead of the less meaningful 16711935. VB has a large number of pre-defined constants for your use, for example *vbRed*, *vbYellow* and so on. The so-called "System Colours" also have constant equivalents, for example, vbDesktop, which represents the colour of your desktop. Refer to the box entitled "Those missing bits…" elsewhere in this article for an explanation of system colours.

VB provides constants for many other things as well as colours, for example, *Window States*, which reflect whether a window is maximised or minimised and so on. In addition, you can define your own constants if you wish – defining your own constants will be covered in a future article

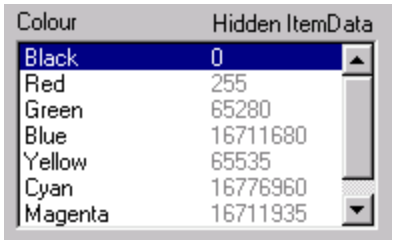
(ED: The filename for this image is “ColourGradient.bmp”)



Varying the levels of red, green and blue can produce the entire spectrum of colours. In this example, only red and green have been used, blue has been omitted. The red intensity has been increased towards the right-hand side of the square and the green intensity has been increased towards the bottom of the square.

Figure 1 – Sample colours using only red and green

(ED: The filename for this image is ItemData.bmp)



The *ItemData* property lets you associate a number with each item in a ListBox. The values you use are never displayed; they are shown here so that you can see what is happening behind the scenes. The values used in this case are the 32-bit RGB values of the colours they represent.

Figure 2 – The *ItemData* property