

C# How-to 6: Use the Windows Form StatusBar control (part 1)

Using the status bar control supplied with Visual Studio .NET is simple; there are a couple of gotchas though if you are used to Visual Basic 6. This page explores the more interesting aspects of the status bar, and how it differs from earlier versions.

1. Design time

Add the control to a form from the toolbox in the usual way. By default, the control is a simple, blank bar which docks to the bottom of the form, although you can alter the docking position and the edges of the form to which it is anchored. Usually though, you'll want to accept the default.

You can use the bar as it is if all you want is a simple line of text, like this:

```
statusBar1.Text = "This is a status bar!";
```

The bar itself has a `Text` property which is only visible as long as the panels are not visible. If they are, they take precedence and the control's own text disappears.

There are two other interesting properties. The `SizingGrip` property allows you to turn on or off the sizing grip - the ridged triangle at the lower right corner of the window which the user can grab to resize the window. You need to be able to turn this off because if you add a status bar to a fixed-size dialog box, you don't want it to look as if the window can be resized.

The other property is the `ContextMenu` property which, as it implies, allows a context menu to be attached to the control which appears if the control is right-clicked. You can add a context menu object from the toolbox and set it up using the regular menu editor, then use the Properties window to attach the menu to the control. This is a good touch.

The control also has a very large number of events, which are mostly standard Windows Forms events (Click, MouseHover, etc.) but one - `PanelClick` - is a bit different. More on this later.

By itself, the control is useful, but not that useful. To make it more so, we need to add some panels to it. You do this by selecting Panels from the Properties window, then clicking the button to the right. This brings up a dialog box which allows you to add, remove, and re-order a series of panels.

Unfortunately the panels themselves aren't very exciting. They can have both an icon and text, and tooltips can be set for each individual panel, which is nice. If you want anything more esoteric you will need to set the `Style` property of the panel to `OwnerDraw`, but then you will have to add code to redraw the panel yourself. This is beyond the scope of this page.

Once you have added one or more panels, you can run your program, and then you find that the panels do not appear. This is the first gotcha - the status bar control has a `ShowPanels` property which is set to `False` by default and which must be set to `True` to display the panels. It took me a while to figure that one out!

The panels themselves do not have any events - well, except the `Dispose` event which is not very useful to us now. This is a little odd because on the face of it you can't even detect when a panel is clicked. However, you can do this because the status bar does have a `PanelClick` event which is fired when a panel is clicked. The handler for this event takes a `StatusBarPanelClickEventHandler` object as a parameter, which contains event data such as the panel which was clicked, the mouse button pressed, the mouse position, etc. But this does mean that `MouseHover` and other useful events are not fired by the panel object.

2. Referring to the status bar panels in code

It's worth saying a little bit about this because there's another gotcha. When you add a status bar plus (say) two panels to the form, the designer adds this code to the form's class:

```
private System.Windows.Forms.StatusBar statusBar1;  
private System.Windows.Forms.StatusBarPanel statusBarPanel1;  
private System.Windows.Forms.StatusBarPanel statusBarPanel1;
```

The designer then adds code to add an array of status bar panels to the status bar. What this means is that you can access the panel directly, rather than having to use the `StatusBarPanels` property of the status bar, which is a collection of all the panels on the bar. So you can access an individual panel like this:

```
statusBarPanel1.Text = "Some text in the panel!";
```

Of course, you can use the `Panels` collection if you wish, and then you would do:

```
statusBar1.Panels[0].Text = "Some text in the panel!";
```

Note a couple of points here. Firstly, according to the documentation the `Panels` collection has an `Item` property which allows you to access the panel object with a given index. So if you were a VB6 programmer moving to C# you might think you could do this:

```
// this won't work!!  
statusBar1.Panels.Item(0).Text = "Some text in the panel!";
```

It won't work because the compiler tells you there is no `Item` property. Secondly, if you use the `Panels` collection directly, you might be tempted to try this:

```
// note parentheses, i.e. Panels(0)  
// this won't work either  
statusBar1.Panels(0).Text = "Some text in the panel!";
```

You have to use square brackets, not parentheses, to enclose the index integer (zero in this case). Why? Because the `Item` property is the indexer for the `Panels` collection. Indexers are a C# construct which I'm not going to explore any further here except to say that they allow you to treat a class as if it were an array, and use the `[]` notation to do so. This also took me a while to figure out, but it does make life easier when you understand how it works.

3. An example

The second part of this tutorial gives a worked example of adding and using a status bar in an application.

C# How-to 6: Use the Windows Form `StatusBar` control (part 2)

This page gives a worked example of how to add and use a status bar control.

Steps to take

1. Example code

As an example, we'll add a status bar to `Simple Editor`, the basic Windows Forms application I created in the first of the C# tutorials on this site. We want the status bar to do two things:

display the current date and time

display the status of the `CapsLock` and `NumLock` keys

If you want to follow along with this, and you haven't already entered the code for the editor so far, you can download the code at its current stage and work from here.

2. Add a status bar to the main form

Assuming you are now looking at the main form of the editor project, add a status bar from the Toolbox to the form. It will automatically dock to the lower edge of the form, which is what we want.

This is going to cause a slight problem because we previously arranged to have the rich text editor box resize automatically when the form was resized. We now have to take into account the height of the status bar, which occupies part of the form's client area. Alter the form's resize event to look like this:

```
private void frmMain_Resize(object sender, System.EventArgs e)
{
    rtfMain.Size = this.ClientSize;
    rtfMain.Height -= statusBar1.Height;
}
```

The second line of the function body adjusts the height of the rich text box appropriately.

3. Add panels to the status bar

Add three panels to the status bar by going to the Properties window and clicking the button next to the Panels line. When the dialog box appears, add three panels with the names panDateTime, panCaps, and panNum. Change the width of the second and third panels to 35 pixels and the alignment to Center. Finally, set the Text property of the second panel to 'CAPS' and that of the third panel to 'NUM'.

The width of the first panel has to accommodate a string of varying length and you could set its AutoSize property to Contents, which would resize it depending on how long the date/time string was. Do that if you wish, but as the date and time change the panel size may change as well, which looks a little odd. If you leave AutoSize as None, then set the width to 265 pixels which should be enough. You can leave the Text property alone.

Leave all other properties at their defaults.

4. Add a Timer control

From the Toolbox, add a Timer control and set the Interval property to 1000 (i.e. 1 second) and Enabled to True. Add an event handler to the Tick event (double-click the Tick event in the Properties window). In the event handler we will update the date and time at 1-second intervals.

5. Add code to the Tick event handler

All we do in the event handler is get the current date and time and display them in the first panel on the status bar. We declare a DateTime structure (this is not a class, despite what I have read in various .NET literature) and initialise it to the current time. Then we use the DateTime structure's methods to get the system short time and system long date as strings:

```
private void timer1_Tick(object sender, System.EventArgs e) {
    // update the date and time in the status bar when the event occurs
    DateTime dtCurrent = DateTime.Now;
    panDateTime.Text = dtCurrent.ToShortTimeString() + " on " +
    dtCurrent.ToLongDateString();
}
```

You can experiment with the properties and methods of the DateTime object to get different results.

If you now run the program, you will see the date and time update at regular intervals (you may or may not see a seconds display depending on the setup for the short time display in your system).

6. Update the CAPS and NUMLOCK key status

For this I am going to use the component I developed in the second C# tutorial on this site. If you don't have this, download the source and compile the component, then customize your toolbox to include the new component. See the tutorial for more details of the component.

Add two instances of the KeyState component to the form (one for the CAPS panel and the other for NUM). Rename them keyStateCaps and keyStateNum, and set the key to monitor as the CapsLock and NumLock keys respectively. Also set the Enabled property for both to be True.

When the program starts, we don't know the status of the Caps and NumLock keys, so we need to get these and set the panel text accordingly. We can do this in the form's constructor, so modify the constructor like so:

```
public frmMain()
{
    // ... rest of constructor as before

    // check initial CapsLock and NumLock key status
    // and modify the status bar accordingly
    if (keyStateCaps.KeyStatus() == false)
        panCaps.Text = ""; // no display if CapsLock is off
    if (keyStateNum.KeyStatus() == false)
        panNum.Text = ""; // no display if NumLock is off
}
```

7. Add event handlers for the keyState components

The last thing we do is add event handlers for each KeyState component and alter the panel text accordingly. The two handlers are almost identical:

```
private void keyStateCaps_KeyChanged(object sender, MbKeyState.KeyChangedEventArgs e) {
    // CapsLock status has changed, so check the new status and update
    // the panel accordingly
    if (keyStateCaps.KeyStatus() == false)
        panCaps.Text = "";
    else
        panCaps.Text = "CAPS";
}
```

```
private void keyStateNum_KeyChanged(object sender, MbKeyState.KeyChangedEventArgs e) {
    // NumLock status has changed, so check the new status and update
    // the panel accordingly
    if (keyStateNum.KeyStatus() == false)
        panNum.Text = "";
    else
        panNum.Text = "NUM";
}
```

The only difference is the panel whose Text property is changed. There are two ways in which this could be improved, if you're interested:

we could use one event handler rather than two and check the sender object to see which one generated the event and it would be nice if the KeyState component returned the new key status to save us checking it manually

Feel free to modify either the component or the handlers if you wish.

That completes a brief look at the StatusBar control. It's basic - but it is easy to use. More functionality is coming Simple Editor's way, so check back for future updates. You can download the complete code from the link below if you don't want to type it in.

[Download code for Simple Editor](#) with status bar (52K) (Note: includes the KeyState component DLL.)