C# Tutorial #2: Developing a Windows Forms component (part 1)

This tutorial shows how to develop a Windows Forms component that can be reused in any Windows Forms program. This isn't the same as a control - but the only difference is that a control draws its own user interface while a component does not. This means that components are used for functions where no visible interface is required.

We want the component to do something useful, so we will write a component that notifies the host program if the status of one of the special keys - CapsLock, NumLock, and ScrollLock - on the keyboard changes. That is, if the key was off and is now on, or vice versa.

The component should have the following properties:

the key to check;

the polling interval (i.e. the interval at which the key status is checked);

and whether the component is enabled or not.

It should also have one event, which will be fired when the status of the relevant key changes, and one method, to allow the user manually to check the key status.

Steps to take

1. Create a new C# project, choosing 'Class Library' when the dialog box appears. I called this project MbKeyState and you might find it better to stick with this as it affects the namespace of the project, which is important for understanding the code later.

2. You now have the skeleton of the project, but there are two other things to do:

change the class name to KeyState (this isn't strictly necessary but otherwise the class and the namespace it is in have the same name, which I find can be confusing)

the class must be derived from the Component class, so you will also need to add the System.Component namespace (add this below the line 'using System')

The class declaration should be changed to look like this:

public class KeyState : Component {

3. We will check the status of these keys by making a Windows API call at regular intervals. The API call is called GetKeyState and takes a parameter which is the internal value of the key (not its ASCII value - these keys don't have one). Rather than use numbers which we might forget, we'll set up an enumeration to make life easier. This should be a member variable of the class and we'll make it public so that users of our component can use it too. The code is:

```
// key type enumeration
public enum KeyID {CapsLock = 0x14, NumLock = 0x90, ScrollLock = 0x91};
```

The hex numbers are the internal key code values.

4. Now we can add some private class member variables. We need internal variables to hold the polling interval, the key we are polling, the key's current status (on or off), and whether the component is enabled or not. The code for these is self-explanatory:

```
// internal variables for property values
```

```
private int m_Interval = 50; // default timer interval
private bool m_Enabled = false; // not enabled by default
private bool m_kStat; // will hold the initial status value
private KeyID m_KeyToCheck = KeyState.KeyID.CapsLock; // check CapsLock by default
```

The default polling interval is 50 milliseconds, which should give an instant response to one of the keys being pressed without too great a performance hit. But if the user is worried about that, we'll let him change it. By default we'll check the CapsLock key. We don't know its status yet - we'll need to check that when the component runs. Finally, the component is disabled by default (you can change this if you like).

5. How are we going to poll the key at regular intervals? For this we'll need a timer. We can add a timer as a class member variable by adding the following code below the other variables:

```
private static System.Timers.Timer m_Timer = new System.Timers.Timer();
```

6. We need a function to interrogate the relevant special key. This should preferably return false if the key is off, and true if it's on. We'll use the API function GetKeyState, but this doesn't return a boolean, which is what we want, so we'll wrap it in our own function like this:

```
private bool GetSpecialKey() {
int nKstat;
// get the key state
nKstat = GetKeyState((int)m_KeyToCheck);
// check the lowest bit to see if the key is on or off
nKstat &= 0x01;
// return a boolean
if (nKstat == 1)
return true;
else
return false;
}
```

Note the cast of m_KeyToCheck to an int, since this is a KeyID enum variable. Don't worry about the API call for the moment - I'll return to that later.

7. Now we need to do some work in the component's constructor. Specifically, we want to:

turn the timer off as we want the component to start disabled;

set the timer's interval to our internal value;

get the initial status of the selected key.

The following code in the constructor will do this:

```
public KeyState() {
// set up the timer
m_Timer.Stop(); // by default we don't start the timer
m_Timer.Interval = m_Interval; // set the timer's interval
m_kStat = GetSpecialKey(); // now we know the key's initial value
}
```

There is one point to note here. We could halt the timer by setting its Enabled property to false. But in the .NET documentation, there is a statement that a disabled timer is susceptible to garbage collection. We really don't want to lose our timer during operation of the component! I don't know if this would really happen - but why risk it?

That completes the project's skeleton. The next phase is to add the properties and the method, and we will do this in part 2 of this tutorial.

C# Tutorial #2: Developing a Windows Forms component (part 2)

In the second part of this tutorial, we will add the necessary properties and method of the component. But first, we need to set up the class so that the Windows API call in our function GetSpecialKey() can be made.

Steps to take

1. To call an API function, we have to assign an attribute to the function name (GetKeyState in this case) to indicate that it comes from an external, unmanaged DLL. First you need to add another namespace, so insert the line 'using System.Runtime.InteropServices;' below the other two namespaces.

2. Next, in the body of the KeyState class, insert the following code (I put it directly after the class declaration):

```
[DllImport("User32.dll")]
public static extern short GetKeyState(int nVirtKey);
```

This tells the compiler that here is an unmanaged function in the DLL 'user.dll' and also gives the declaration for the function. Now the compiler won't flag an error when it sees a call to this function in the KeyState class. The parameter to the function indicates which key is to be interrogated.

3. Now we will add some properties to the component. Before we do though, ideally we would like these to show up in the property browser in the VS.NET interface, just like the properties of the components supplied with .NET. To do this we use design-time attributes, which allow us (among other things) to specify which category of the property browser the property should appear in, and a description to be displayed when the property is clicked on.

Adding an Interval property will show how this is done. All the property function does is to return or set the value of the internal variable m_Interval, like so:

```
[
Category("Behavior"),
Description("The number of milliseconds in between checking the special key status."),
]
public int Interval {
get { return m_Interval; }
set { m_Interval = value; }
}
```

Which is pretty straightforward. The design-time attributes come immediately before the property definition. They simply specify the property browser category and the property description. The syntax is simple. There are quite a lot of these attributes and they can be found in the .NET documentation by searching under 'Design-Time Attributes for Components'.

4. We can add Enabled and KeyToCheck properties in the same way:

```
[
Category("Behavior"),
Description("If True, the component will signal changes in the special key status."),
]
```

```
public bool Enabled {
get { return m_Enabled; }
set {
m_Enabled = value;
// start or stop the timer depending on Enabled property
if (m_Enabled == true) {
m_kStat = GetSpecialKey(); // get the key's current status
m_Timer.Start();
}
else
m_Timer.Stop();
}
}

[
Category("Behavior"),
Description("Identifies which key status is to be checked"),
]
public KeyID KeyToCheck {
get { return m_KeyToCheck; }
set {
m_KeyToCheck = value;
// get the new key's current status
m_kStat = GetSpecialKey();
}
}
```

5. Finally, here is the simple method to allow the user to query a key's value manually (note that this works even if the component is not enabled - all the Enabled property does is turn on or off the event which signals a change in key status):

```
public bool KeyStatus() {
return GetSpecialKey();
}
```

This method does not indicate which key has been tested - it's up to the host program to keep track of that, but this is available through the KeyToCheck property if required.

That completes the adding of properties and methods. The last thing to do is to add an event which will fire when the key's status changes. That requires a page all to itself, which is part 3 of this tutorial.

C# Tutorial #2: Developing a Windows Forms component (part 3)

In the third part of this tutorial, we will add code to raise an event when the status of the relevant special key changes. Although this looks complex, it is relatively straightforward once you get the idea of how it works.

Steps to take

1. In the pages on this site dealing with event handlers, it was noted that a function to be used as an event handler has to match the signature of a delegate function which is delegated to handle the event.

This means that in our code we must supply that delegate signature. Also, the event handler will have two parameters: the object which raised the event, and an object containing information passed to the handler and

which is derived from the .NET framework class System.EventArgs. We should therefore supply this, too. We also need to declare an event, and finally, we need a function which actually raises the event.

2. To tie all this together in the VS.NET environment, a strict set of naming rules have to be followed (if you're not using VS.NET to write your programs, this probably doesn't matter - but if you are, follow the convention).

These rules are:

the delegate must take the form EventNameEventHandler;

the class derived from System.EventArgs must be named EventNameEventArgs;

and the function which raises the event must be named OnEventName.

We will call the event 'KeyChanged' so we need a delegate definition named KeyChangedEventHandler, a class named KeyChangedEventArgs, and a function named OnKeyChanged. If we do all this, VS.NET will tie the various parts together and enable anyone using the component to do things like double-clicking on the event name in the property browser and having VS.NET automatically insert an event handler of the correct form.

3. First, we'll add the new class. You could do this in the main code file, but it keeps things tidier (to my mind, anyway) to create a separate file. However you do it, you need to end up with something like this:

```
using System;
namespace MbKeyState
{
/// <summary>
/// This class contains the data made available to the component's
/// container when the KeyStateChanged event is raised.
/// </summary>
public class KeyChangedEventArgs : System.EventArgs {
public KeyChangedEventArgs() { } // constructor (does nothing)
}
}
```

Now, you'll note that this class doesn't actually do anything, and in fact we needn't have done this - because we're not passing any custom data, we could just have used a new System.EventArgs object. But if we ever do want to add some custom data, we have this class ready.

Important: however you add the new class, make sure it is in the same namespace as the original class!

4. Next, to add the delegate definition. This should come within the namespace MbKeyState but outside the KeyState class. The definition is simple:

```
public delegate void KeyChangedEventHandler(object sender, KeyChangedEventArgs e);
```

Note that the second parameter is our new KeyChangedEventArgs class.

5. The next step is to declare the event. We do this in the body of the KeyState class, as the event we declare will be raised by that class. The event declaration is:

```
[
Category("Behavior"),
Description("Event fired when the status of the special key changes"),
]
```

```
public event KeyChangedEventHandler KeyChanged;
```

So the event is called KeyChanged and any handler must be of the type KeyChangedEventHandler, which we defined in step 4. Note that, just as with the properties of the component, we use a design-time attribute to add some information about the event which will appear in the property browser.

6. Lastly, we need a function in the body of class KeyState to raise the event. This is simple and consists of the following:

```
protected virtual void OnKeyChanged(KeyChangedEventArgs e) {
if (KeyChanged != null)
KeyChanged(this, e);
}
```

So there we are. The function OnKeyChanged raises the previously declared event KeyChanged with a parameter of the calling object ('this') and a KeyChangedEventArgs object. All done.

Well, not quite. When is the function OnKeyChanged actually called? And where does it get its KeyChangedEventArgs parameter from?

7. We only want OnKeyChanged to be called (and thereby raise an event) when the special key we are monitoring is toggled on or off. We included a timer object to monitor the key status, which means we need to add an event handler for the timer which is called when the interval has elapsed.

Unfortunately there is no visible interface to this component and therefore we can't add an Elapsed event handler by double-clicking the event in the property browser, since the browser is not available to us! So we will have to do this manually.

The timer event handler will poll the key status every time it is called, and compare it with the previous status. If it has changed, the handler will call OnKeyChanged. The delegate signature for the timer class's Elapsed event takes an object of type System.Timers.ElapsedEventArgs, so the complete function looks like this:

```
private void m_Timer_Elapsed(object sender, System.Timers.ElapsedEventArgs elap) {
bool bNewStat;
// check the key's status
bNewStat = GetSpecialKey();
// fire event if status changed
if (bNewStat != m_kStat) {
m_kStat = bNewStat; // keep new value of key status
KeyChangedEventArgs ek = new KeyChangedEventArgs();
OnKeyChanged(ek);
}
}
```

I have named this function m_Timer_Elapsed as this is the name that the VS.NET designer would have used if we had been able to insert the handler by double-clicking the event name. Note that we create a new KeyChangedEventArgs object and use this as the parameter in the call to OnKeyChanged.

8. The final step is to link this handler to the timer's Elapsed event. Normally, the VS.NET designer would have done this for us, but we have to do it manually. I chose to do this in the KeyState class constructor, so add the following line to the constructor after those already there:

```
m_Timer.Elapsed += new System.Timers.ElapsedEventHandler(m_Timer_Elapsed);
```

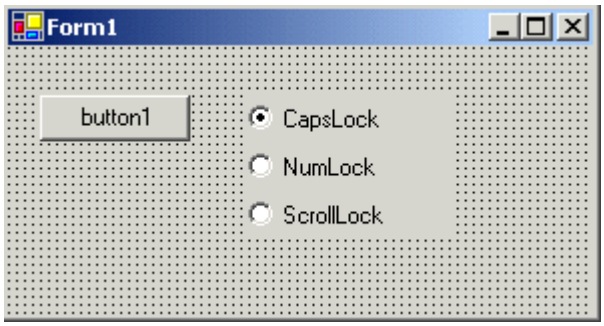
All this does is add a new event handler of the correct type to the event and points to our handler function as the one to add (the clear implication here is that we could add more than one handler - which could be interesting...).

The component is now finished and ready to go. We will have to create a test container to try it out, and this is demonstrated in part 4 of this tutorial.

This is the last part of the tutorial and simply shows how to test the component in a container.

Steps to take

1. Close the component project and start a new Windows Forms project. Add to the form a button plus three radio buttons. Change the caption of each radio button to match one of the keys, and set the CapsLock radio button to be checked. The result should look something like this:



2. Go to the Tools menu in VS.NET and click Customize toolbox. We do this to make our new component available to the application. When the dialog box eventually opens, click the '.NET Framework Components' tab and when the machine stops grinding away, click the Browse... button. Navigate to the built component (should be in the \bin\debug folder of the KeyState project folder) and double-click KeyState.dll. You should find a new component - KeyState - in the list; ensure that the box to the left of its name is checked, and click OK.

3. The new component will be in the General section of the toolbox, and will have a cogwheel icon since we haven't given it anything else. Click and drag the component to the form; when you drop the component, notice it ends up in a separate section below the form since the component has no visible interface and therefore doesn't need to appear on the form (this is different from VB6 where all components, visible or not, ended up on the form).

4. Click the component, which should have the name keyState1, and in the property browser set the Enabled property to True. Note that the KeyToCheck property is set to CapsLock, and the Interval to 50 (milliseconds); these are the default values we gave the component when it initialised. Also, note that the properties fall in to the category we assigned with the design-time attributes, and that the descriptions we gave appear in the browser when the property is clicked.

5. Now we can add an event handler for the component. Switch to the event pane of the property browser and double-click the KeyChanged event. In the code window you should get this:

```
private void keyState1_KeyChanged(object sender, MbKeyState.KeyChangedEventArgs e) {

}
```

Note that the event handler has the correct signature as specified in our component. We can add some code to this handler to indicate when the event occurs:

```
if (keyState1.KeyToCheck == MbKeyState.KeyState.KeyID.CapsLock)
```

```
MessageBox.Show("The CapsLock key has changed!");
else
MessageBox.Show("One of the other special keys has changed!");
```

Obviously the code in a real application would do something more useful, but you can see how this handler is called only when the key being monitored changes, and how it reads the KeyToCheck property of the component to see which key that is (if we didn't already know!).

6. Add a Click event handler to the single button and insert the code as follows:

```
private void button1_Click(object sender, System.EventArgs e) {
// interrogate key status
if (keyState1.KeyStatus()==false)
MessageBox.Show("The monitored special key is off");
else
MessageBox.Show("The monitord special key is on");
}
```

This just tests the KeyStatus method of the component.

7. Finally, add an event handler to the CheckedChanged event for each radio button, and add the code in the lines below:

```
private void radioButton1_CheckedChanged(object sender, System.EventArgs e) {
if (radioButton1.Checked==true)
keyState1.KeyToCheck=MbKeyState.KeyState.KeyID.CapsLock;
}
private void radioButton2_CheckedChanged(object sender, System.EventArgs e) {
if (radioButton2.Checked==true)
keyState1.KeyToCheck=MbKeyState.KeyState.KeyID.NumLock;
}
private void radioButton3_CheckedChanged(object sender, System.EventArgs e) {
if (radioButton3.Checked==true)
keyState1.KeyToCheck=MbKeyState.KeyState.KeyID.ScrollLock;
}
```

All this does is change the key which we are monitoring. You can use this to show that the KeyChanged event is only raised when the correct key is pressed.

That completes the component tutorial. It is quite easy to develop these components once you get the hang of it, and they really allow you to encapsulate some useful code. If you want to download the files to build the component and the container, click the links below.

Download code for KeyState component (21K)
Download code for test container program (41K)