

Determining Form and Process Changes in Windows CE

August 2008
Christopher Tacke
OpenNETCF Consulting, LLC

Introduction

Windows CE, and especially Windows Mobile devices have a UI paradigm that's quite a bit different than the desktop. As a general rule, only one application, and indeed only one Form is typically ever in front of all other applications because there usually isn't enough real estate to show multiple forms and MDI layout isn't supported in CE.

One of the things that this paradigm pushes us toward is knowing either when the visible Form changes or when the user changes applications completely. We can use this for tracking state or even implementing an application authentication scheme.

Unfortunately the Windows CE operating system simply doesn't provide us any kind of event to tell us when a Form has changed or when the process has changed. Sure, within our application we can watch things like the Deactivate event on a Form, but when our Form gets deactivated, how do we determine what's on top now? Is it another Form in our application, or another application altogether?

In this article we'll take a look at a simple utility class called WindowWatcher than can provide us feedback for these state changes through a very simple managed event interface.

Window Handles and Process IDs

The key to our solution is the fact that all Forms are Window and all Windows have a unique identifier called a Handle, or hWnd, and that all processes have a unique identifier called a process identifier, or PID. There are Win32 APIs to retrieve these identifiers, and while there are no corresponding managed-code methods that call them, a little P/Invoke gets us there easily.

The Win32 APIs we're going to be interested in are:

Win32 Method	Function
GetForegroundWindow	Gets the handle, or hWnd, of the current topmost, visible Window
GetCurrentProcessId	Gets the process identifier, or PID, of the calling process
GetWindowThreadProcessId	Gets the PID that owns a specific hWnd

With just these three APIs we can determine when the foreground Window has changed because the return value from GetForegroundWindow will change, and we can determine if the new foreground Window is in a new process because the PID retrieved from GetWindowThreadProcessId will change.

A Simple Solution

So armed with these APIs, and a general idea of how we can determine the state changes we want, we can create a simple class that encapsulates our logic.

First we need to define a couple delegates for our events:

```
public delegate void WindowChangedHandler(bool processChanged, bool inMyProcess);
```

An event of this type will be raised any time the foreground Window changes, and if the owner of the new Window is a new process, processChanged will be true. In either case inMyProcess will tell us if the new foreground Window is in the process that created our WindowWatcher class.

Now we implement the logic. The core of the logic will be a background thread that periodically checks the current foreground window. To be nice, we'll make the interval adjustable.

```
public int Interval { get; set; }
```

Next we define some variables that we'll maintain at class scope to keep track of state between checks and initialize them in the class constructor plus define the event of our delegate type:

```
private bool m_enabled;  
private bool m_stopThread;  
private IntPtr m_lastForeWindow;  
private uint m_lastPID;  
private uint m_myPID;  
  
public event WindowChangedHandler WindowChanged;  
  
public WindowWatcher()  
{  
    m_enabled = false;  
    m_lastForeWindow = IntPtr.Zero;  
    m_lastPID = 0;  
    // call the SDF because it's not a P/Invoke - it's a kcall and this is less code  
    m_myPID = (uint)OpenNETCF.Diagnostics.ProcessHelper.GetCurrentProcessID();  
  
    Interval = 100;  
}
```

You'll notice here that to get the current process ID I'm not just calling a P/Invoke, but instead calling into the Smart Device Framework (SDF). The reason for this is that GetCurrentProcessID isn't a true API, but is instead a macro that's defined in the CE headers, and the value it returns depends on the processor architecture of the device. Rather than lay out all of the code needed to determine this it's a lot easier to just call a function in the SDF. Keeps the code smaller and less confusing.

While we're thinking about P/Invokes, we might as well define the other two we do need:

```
[DllImport("coredll.dll", SetLastError = true)]  
private static extern IntPtr GetForegroundWindow();  
  
[DllImport("coredll.dll", SetLastError = true)]
```

```
public static extern uint GetWindowThreadProcessId(
    IntPtr hwnd, out uint lpdwProcessId);
```

To start and stop our watcher, we'll expose a simple Enabled property. It's job will be to start or stop our background worker thread.

```
public bool Enabled
{
    get { return m_enabled; }
    set
    {
        if (value == m_enabled) return;

        if (value)
        {
            Thread thread = new Thread(WatcherThreadProc);
            thread.IsBackground = true;
            thread.Start();
        }
        else
        {
            m_stopThread = true;
        }
    }
}
```

The only thing remaining is to implement the actual logic engine of this class that does all of the work, which is all in a single method running as the background thread:

```
private void WatcherThreadProc()
{
    // initialize some values
    m_stopThread = false;
    m_lastForeWindow = GetForegroundWindow();
    GetWindowThreadProcessId(m_lastForeWindow, out m_lastPID);

    while (!m_stopThread)
    {
        IntPtr currentForeWindow = GetForegroundWindow();

        if (currentForeWindow != m_lastForeWindow)
        {
            uint pid;
            GetWindowThreadProcessId(currentForeWindow, out pid);

            // window has changed - is it a new process?
            bool newProcess = false;
            if (pid != m_lastPID)
            {
                newProcess = true;
                m_lastPID = pid;
            }

            bool inMyProcess = (pid == m_myPID);

            if (WindowChanged != null)
            {
                WindowChanged(newProcess, inMyProcess);
            }

            m_lastForeWindow = currentForeWindow;
        }
    }
}
```

```
Thread.Sleep(Interval);
}
m_enabled = false;
}
```

You can see that it runs on the period determined by the Interval property. Ever Interval, it checks the foreground Window. If it has changed, it then determines if the new foreground Window is in the same process as the last. Next it determines if the process that owns the foreground Window is the same process that created the watcher class itself and finally it fires off an event if there are any subscribers.

That’s all there is to it. Using the new class is very, very simple. It looks like this:

```
WindowWatcher watcher = new WindowWatcher();
watcher.WindowChanged += WindowChangedHandler;
...
void WindowChangedHandler(bool processChanged, bool inMyProcess)
{
    if(processChanged)
    {
        Debug.WriteLine(string.Format(
            "Window and Process changed - we {0} in our app",
            inMyProcess ? "are" : "are not"));
    }
    else
    {
        Debug.WriteLine(string.Format(
            "Window changed - we {0} in our app",
            inMyProcess ? "are" : "are not"));
    }
}
```

Conclusion

As Compact Framework developers, and really even as general mobile and embedded device developers, we have to get used to the fact that Microsoft can’t reasonably think of every possible use-case or business challenge we might face. What’s important is that they’ve generally provided us the tools to implement solutions to just about any problem we run into.

In this case we needed to know information about not just application state, but overall device state, and in less than 100 lines of code we came up with a reasonable solution that didn’t require any onerous coding or framework that we’d have to inject into all of our Forms. As with many problems like this, understanding the Win32 API and thinking outside the managed-code box can usually get you a fairly simple solution.