

Power Aware - Save the Batteries

by John Robbins

I recently attended a Microsoft Tablet PC Dev Lab in Redmond. While there were lots of very interesting discussions on future software and hardware, nearly every session had someone saying something like, "All this is great, but the big problem is still power." No matter how cool the future looks, something tells me I'm still going to be running around airports, cafés, and meeting rooms desperately looking for an electrical outlet to recharge my machine's battery. Certainly, battery technology is not subject to Moore's Law.

I started thinking what I could do as a software developer to be more cognizant of power consumption when the machine is running on battery. This is something most developers don't think about much. I know that until I went to Dev Lab, I had never really thought about things like checking if the disk was powered down so my program wouldn't cause it to power back up. The LCD chews up half the power when running on batteries; the software you and I write normally doesn't use that much overall. Any little bit helps, though.

My initial thought was that the .NET Framework Class Library (FCL) already had code in there to help you with smart battery usage. Sadly, there's essentially no support at all. So I wrote a set of classes that you can use to put some power smarts into your applications. Before I discuss into the classes, I want to examine the power support in the FCL and Win32 (where all life originates for .NET applications), so you'll know what's supported where. If we all start planning our applications to run on battery power, maybe we'll be able to actually use our computers on flights all the way across the ocean!

The Win32 and FCL Support for Power

A bunch of years ago, I worked on an application that needed to know a little bit about the state of the power system. So when writing these new classes, I knew I should start with the **WM_POWERBROADCAST** message. If you search for it on the MSDN Library or MSDN Online, you'll see enough messages and functions to find out more than you ever wanted to know about power. Quite a bit of it is related to handling power schemes you set with the Power Control Panel applet. The real keys to battery power are the functions such as **GetSystemPowerStatus** and **GetDevicePowerState**.

After familiarizing myself with the base operating system capabilities, I started poking through the FCL for its power support. As I mentioned, there is basically none. While the **SystemEvents** class offers the **PowerModeChanged** event, it's much too limited for real world use. The **PowerModeChangedEventArgs** will tell you through its **Modes** property will tell you the system suspended, resumed, or had a power status change, but it won't tell you what you really want to know: have you transitioned from AC power to battery.

You can handle the status changes yourself and manually grovel through the Win32 API for why you're getting them, but that seemed pretty pointless. I immediately saw that I needed to provide three notifications that your application can actually do something about: when the battery has reached a critical point, when the machine is running on battery power, and when the machine goes on AC power.

Of course, to garner all this information, I would need to wrap the important Win32 API power functions and structures. Before I get into the implementation details, I want to discuss how to use the library I put together. Once that's out of the way, I'll discuss the tricks I used to get everything working.

Using Wintellect.Win32

The meat of Wintellect.Win32 is the PowerEvents class. I modeled it on the existing SystemEvents class, with the difference being that PowerEvents solely focuses on serving up power events. The first event is PowerModeLowBattery, which, as the name hints, will trigger when the machine is about to run out of juice. The event arguments passed in will tell you the percentage of the battery that's left. If you get this event, you'll want to ensure that your application is prepared to shut down as the machine will probably be doing just that in a few minutes.

The other two events served up by the PowerEvents class are somewhat codependent: PowerModeOnACPower and PowerModeOnBatteryPower. These are triggered to let you know when the machine moves on and off the two states. Now you'll easily know when you're running on the battery versus the power plug. When you get the PowerModeOnBatteryPower notification, you can avoid hitting the disk or know that you'll probably have a step throttling CPU which will run slower on battery. For example, one application I am working on automatically saves the user data much like Microsoft's **OneNote**. In the options, I allow multiple automatic save times for the user, one when running on power and one when on battery. That way they have the option to keep the auto save interval the same, but can extend it to save battery power if they desire

On nearly all systems I tested the code on, there's a one-to-one relationship between the PowerModeOnACPower and PowerModeOnBatteryPower events. On some machines, switching from AC power to battery power actually is internally a three-step operation: reports that the machine is on battery, reports that there's no battery in the system so it's going on AC power, and finally reports that the machine is on battery power. On those machines, the PowerEvents will report what the machine is telling it. In your code, you may want to keep the time of the event because you'll see the PowerModeOnBatteryPower event followed by the PowerModeOnACPower event in the same second. One second later, you'll see the final PowerModeOnBatteryPower event.

If you are interested in receiving suspend and resume power notifications, you can. The PowerModeChanged event is simply a wrapper on the FCL's SystemEvent.PowerModeChanged. While many applications can use the PowerEvents class for all their needs, some may need more information about the power state.

If you want the current power status so you can see how much time on the battery is left or if the battery is charging, the PowerStatus class fits the bill. You can either allocate a new class every time you want to check the current status, or call the Update method on any previously allocated PowerStatus variable.

The `ProcessorPowerInformationArray` returns an array containing individual `ProcessorPowerInformation` classes for each processor. This means you can get information such as how fast the processors are currently running and what their idle state currently is. While I could have simply returned the power status for just the first processor on the machine, I thought I should plan for the day when our laptops have dual processors on them.

Some applications, such as a video player or an application that absolutely must complete an operation before suspending, need a way to tell the operating system to keep the display from blanking out or resetting the internal suspension timers. The `SetExecutionState` and `GetExecutionState` methods of the `ExecutionState` class wrap the Win32 API functions for controlling the machine.

The final class in `Wintellect.Win32` is `DevicePowerState`. For really power-aware applications, they can check if a device the app wants to use has power. Since spinning up a disk or other device can be expensive, the methods in `DevicePowerState` can save a good bit of battery power for the user. The `IsSpecificDiskDevicePowered` method takes a `FileStream` and will report if the disk drive for that file is powered. The `IsSpecificNetworkDevicePowered` method does the same if you have a `Socket` class. For devices other than disks and network items, the `IsDevicePowered` method takes a Win32 handle to the device or an object on that device.

To see all the classes in `Wintellect.Win32` in action, the `Test` directory contains a program called `PowerEventsTester`. Considering there is nothing earth-shattering about `Wintellect.Win32`, I'm surprised that a similar library isn't part of the FCL. Now that you have an idea how to use `Wintellect.Win32`, I want to discuss how I got things working.

Implementation Highlights

I started by exploring how I was going to implement the events in the `PowerEvents` class, since they're the meat of `Wintellect.Win32`. If you've looked at the power management code in the Platform SDK, you'll see that the **WM_POWERBROADCAST** message is where the action is. All power state changes come through the message, which means you must have a window handle to receive them. Originally, I thought that all of this would be part of a Windows Forms application, so I would just need to override the `WndProc` method, put in a few handlers, and `I'd` be set. Of course, I'd completely forgotten about possible console applications or Win32 services. Pulling up Lutz Roeder's amazing **Reflector**, I took a look at the decompilation of the `SystemEvents` class to see what it was doing. As I suspected, it creates a hidden window in order to handle the **WM_POWERBROADCAST** message. As there was already a hidden window there for the application to use, which handles other message-based change notifications such as **WM_FONTCHANGE** and **WM_DISPLAYCHANGE**, I didn't think creating a separate window was the way to go. Consequently, I decided to handle the `SystemEvents.PowerModeChanged` event and build smarts into how I handled that notification. As you can see from the code, that turned out to be quite simple.

When I develop code, I always do the unit test first so I can test as I go along. I've been doing this for over a decade; it makes a huge difference. (I sure wish I was smart enough to come up with the fancy term "Test Driven Development!") Anyway, when I started thinking though how I was going to test, I didn't relish the idea of plugging and unplugging my laptop to get the appropriate events generated. Sitting around for many hours waiting for the battery to drain down to a critical state to test my `PowerEvents.PowerModeLowBattery` event wasn't the best way to test.

Since all the power change notifications come to the application as a `WM_` message, all I needed to do was to get the hidden window that the underlying `SystemEvents` class created and pump fake messages to it. A quick look in **Reflector** showed that the private method, `SystemEvents.CreateBroadcastWindow`, generated the name from the following piece of code:

```
SystemEvents.className = string.Format ( ".NET-  
BroadcastEventWindow.{0}.{1}" ,  
    "1.0.5000.0" ,  
    Convert.ToString(AppDomain.CurrentDomain.GetHashCode()),  
    16));
```

Using the same code in the `PowerEventsTester` program, I created nine buttons that push a **WM_POWERBROADCAST** message to the hidden window to duplicate each of the possible power states. Of course, the hardcoded `1.0.5000.0` will probably change for the Framework 2.0, but if `PowerEventsTester` can't find the hidden window, it disables the buttons as appropriate.

In testing my code I noticed something interesting about the **WM_POWERBROADCAST** message. When the power status changes, sub message **PBT_APMPOWERSTATUSCHANGE**, you receive many more notifications than indicated in the documentation. The documentation says "This event can occur when battery life drops to less than 5 minutes, or when the percentage of battery life drops below 10 percent, or if the battery life changes by 3 percent." From my experimentation, the power status change messages come in about every two to three seconds and, most interestingly, whenever another program calls **GetSystemPowerStatus**. I wanted to point this out because if you are going handle the `PowerEvents.PowerModeChanged` event, which is a wrapper on **SystemEvents.PowerModeChanged**, you'll probably want to ignore the **PowerModes.StatusChange** notifications and only handle the suspend and resume notifications if you need them.

The only problem I had developing `Wintellect.Win32` was my own stupidity. To get the processor power information, you use the **CallNtPowerInformation** Win32 function. In the long tradition of Windows, it's one of those functions where you have to pass in all sorts of different-sized buffers and flags to tell the function what data you want. Checking the wonderful www.pinvoke.net, I saw **CallNtPowerInformation** wasn't listed, so I needed to do my own `DllImport` declaration. Lifting the declaration from the documentation, I changed the `ULONG` parameters to `long`s and the `PVOID` parameters to `IntPtr` types.



Unfortunately, I could never get any data out of **CalIntPowerInformation**, but it always returned zero indicating success.

Being completely flummoxed as to what was going on, I spent half a day fiddling and desperately trying to figure out what was going on. In a last effort to see what was wrong, I resorted to stepping through the setup and call to **CalIntPowerInformation** at the x86 assembly language level. Once thing I've learned in this business is that assembly never lies! As I stepped through I was quite confused to see that there were seven parameters being pushed on the stack, when there were only supposed to be five. If you're smarter than I am, you probably realized that in .NET a long is eight bytes, not four bytes as it is in native C++. Sometimes brain cramps hit the best of us.

Wrap Up

If you've ever used a tool like **FileMon** when your machine is running on batteries, it can get a little depressing to see how much disk hitting is going on. My goal for **Wintellect.Win32** was to make it easier to get the power events that you really wanted as well as to provide ways to get the power information itself. My hope is that in the future, programs like **Windows Logo** will make power awareness part of the requirements. Armed with **Wintellect.Win32** and its power classes, you will now be able to properly run when the machine is on batteries so we can all have longer running machines.