

# Windows Mobile Power Management

---

**Joel Ivory Johnson**

*Software Engineer, RDA Corporation*

[jjohnson@rdacorp.com](mailto:jjohnson@rdacorp.com)

## Introduction

As a regular reader of the MSDN support forums and other online development community I see several questions (some reoccurring) about aspects of power management and control on Windows mobile devices. Sometimes a developer wants to reduce the demand on power and other times disable the power saving features. Then there are common coding patterns that contribute to killing the battery much faster. In response to the questions of the community I've collected information on Windows Mobile power management and have compiled it into this document. While Windows Mobile Standard devices are mentioned within this document it is centered on Windows Mobile Professional, though much of this information is applicable to many other Windows CE devices.

## About the Code

The code examples included with this article are a collection of small programs that demonstrate one or two power management concepts each. These programs rely on several native calls. Rather than declare P/Invokes in each individual program there is a single project that contains all of the necessary native calls and the example programs make their low level calls through the classes in that project. I would normally wrap these calls in a helper class but have avoided using one for these examples as not to add a layer of separate between the calls that I wish to demonstrate. All of the sample code is in C# with P/Invoke calls. Since the APIs used are listed the information should be useful to C++ developers as well.

The example programs in the attached code illustrate the following:

- Displaying the voltage, current, and temperature of your device's battery
- Changing the power state of hardware within the device
- Enumerating the hardware in a Windows Mobile Professional device
- Enumerating the power modes that a Windows Professional device supports
- Preventing a device from sleeping
- Toggling the state of the screen's backlight
- Waking up the machine to perform work without alerting the user or turning on the screen
- Keep the GPS hardware powered in sleep/unattended mode

## Driver Dependencies

As mentioned, some aspects of power management deal with low level calls. As such some code examples may not work on your device because of dependencies on the driver implementation. During the course of collecting this information I performed a firmware update on a Windows Mobile 6 device I own (now a 6.1 device). Some examples stopped working and the cause of the problem was that the manufacturer of this specific device decided not to use the standard power manager in their firmware update. Since Windows Mobile is a modular operating system an OEM can choose whether or not to include certain functionality. Many of the code examples will run on Windows Mobile 5 devices, but I am targeting Windows Mobile 6.

## Terms – Windows Mobile Professional and Windows Mobile Standard

With the release of Windows Mobile 5 Microsoft changed their naming of the devices on which Windows Mobile runs from Pocket PC to Windows Mobile Professional and from Smartphone to Windows Mobile Standard (there are also the non-phone Pocket PC devices which are now called Windows Mobile Classic devices. While I do not refer to the Windows Mobile Classic devices from this point anything that I state to be applicable to Windows Mobile Professional is also applicable to Windows Mobile Classic). For the rest of this article I will refer to these devices as Professional or Standard devices. While both devices are available in several different form factors the attribute that can be used to distinguish a Professional device from a Standard device is whether or not it has a touch screen. All Professional devices have touch screens and Standard devices do not. There are other differences in the user interfaces of the two platforms (ex: The start button is on the top of the screen in a Professional device and on the bottom of the screen in a standard device) but recognizing whether or not it responds to touch is usually the easiest attribute to use.

## What does “Off” Mean?

Foundational to this discussion is an understanding of what is mean by the word “off.” A typical light bulb on a typical light switch is either on or off with no states in between. Historically that could be said of many electrical devices, but in today’s world many devices do not frequently use true off states unless their power supply is disrupted. The off state has been displaced by a suspended state or low power state. The power behavior on Standard and Professional devices are different so I’ll describe these separately. But common amongst both platforms is that if I ever say that the device is off that means that power is not flowing through the devices while when a typical user says “Off” they will mean that the devices screen is off irrespective to whether or not the device is actually consuming power or performing actions.

## Windows Mobile Standard

Windows Mobile Standard devices are typically on. If you turn the phone to it’s true off state then you won’t be able to receive phone calls. If you don’t interact with the device the screen will power off and the processor will run at a lower speed but it is still on and running. According to the Windows Mobile Team blog having the device in an always on state contributes to better power savings. A potential problem with this mode is that a badly

written program will cause the processor to consumer more power than it needs to when it is idle.

## Windows Mobile Professional

Windows Mobile Professional has four power modes that we should know about. In the full powered state the device’s screen and backlight are on. If you press the power button or don’t interact with the device then after so much time it goes into the “unattended” state. In this state the device is still running but the screen and backlight are powered down. The user would look at this is being an off state but the device is still awake. After about 15 seconds the device will go from the unattended state to the suspended state. In the suspended state all of the programs that had been running are still in memory but they make no progress since the CPU is in a halted state. The final state we are concerned about is the true off state.

## The Power States

### System Power States

Windows Mobile devices have several predefined power states. Some of these states are specific to the Professional, some to the Standard, and others can be found on both. The following table lists the power states and describes what occurs to transition the device into said power state. The P means that a state is applicable to Professional device and the S means the state is applicable to Standard devices.

State Name		Description
<b>On</b>	SP	Full powered on state
<b>BacklightOff</b>	SP	The user has not interacted with the system for some time and the backlight has turned off. The timeout for this to occur is set through the backlight setting in the control panel
<b>UserIdle</b>	S	This state is currently specific to Standard devices but will be implemented on Professional devices once they switch to the always on power model. In this state the backlight and the screen are turned off.
<b>ScreenOff</b>	SP	In this state the user has specifically turned the screen off. This is different from the screen being ruined off due to idleness.
<b>Unattended</b>	P	This mode is specific to Professional devices. The screen, backlight, and the audio are all turned off but the programs on the device are running. This enables ActiveSync to check e-mail without alerting the user. From the user’s perspective the device is sleeping.
<b>Resuming</b>	P	When a device wakes up from sleeping it is in this mode. The backlight and the screen are still off and programs have a 15 second window to switch the device to another power state before the device automatically goes back to the suspended state.
<b>Suspended</b>	P	The device is sleeping and the processor is halted. The device will not come out of this state until a hardware event wakes it up.
<b>Off</b>	SP	The device is not performing any actions and is not consuming power (exception: though in some devices the system clock may still draw power to maintain time).

Note that programs can alter a devices transition among power states. An example of such a program is an instant messenger which may prevent a device from suspending so that it can continue to receive messages.

## Device Driver Power States

The individual components of Windows Mobile Devices can have their own power states. The most noticeable of which is the screen and the backlight. The device can be on while the backlight or the screen is turned off (a common power configuration when using a media player). The power states for device drivers are named a little more abstractly.

State	Description
<b>D0</b>	Device is fully powered
<b>D1</b>	Device is functional but in power savings mode
<b>D2</b>	Device is in standby
<b>D3</b>	Device is in sleep mode
<b>D4</b>	Device is Unpowered

Depending on the hardware and driver some of these power states are identical. When the power state of a device needs to be changed a request to change the state should be passed through DevicePowerNotify. Requesting a power state change doesn't guarantee that the state will be changed. The driver is still able to make its own decision on whether or not the power state will change (remember your program is not the only program running and isn't the only program that influences power state).

## Requesting and Changing the Power State

Use the native function [GetDevicePower](#) if you need to monitor query the power state instead of polling the state though [GetDevicePower](#) your program can request notification for power change events through [RequestPowerNotifications](#).

If your application needs for the Windows Mobile device to stay in a certain power state then use [SetPowerRequirement](#) to request the power state that your application needs. The OS will ensure that the power state of the device does not drop below what was requested. When the power state is no longer needed then call [ReleasePowerRequirement](#). To set the power state of a device driver instead of expressing minimum requirements use [SetDevicePower](#).

The power state of the system can be set through the native function [SetSystemPowerState](#). The power state for individual hardware items in a Windows Mobile device can be set using [SetDevicePower](#).

(You can also request that your program be started automatically during certain power change events. See <http://www.codeproject.com/script/Articles/Article.aspx?aid=27917> for more details.)

When you have an application that needs to run in unattended mode use the native function [PowerPolicyNotify\(\)](#) to inform the OS that you need for the device to continue to run without suspending. This is a reference-count API. For each call to this function with TRUE you must

also call this function with FALSE to ensure the device does not unnecessarily stay in a higher power.

## Without the Power Manager

While Windows Mobile devices have use of the Power Manager, Windows CE devices may or may not. For devices that do not support the power manager API, calls to the GWES system can be used to power the system down programmatically. Only use GWES functions if the power manager API is not available. [GwesPowerOffSystem](#) suspends the system. Alternatively one can also simulate the pressing of the power button by generating keyboard messages with the `keybd_event` function as follows:

```
keybd_event( VK_OFF, 0, KEYEVENTF_SILENT, 0);  
keybd_event( VK_OFF, 0, KEYEVENTF_SILENT|KEYEVENTF_KEYUP, 0);
```

In either case since the messages must be processed by GWES the function calls will return and a few more lines of code will execute before the system suspends. If you need for your system to stop doing work after calling this method then make a call to [Sleep](#) .

## Enumerating Hardware

Active hardware in Windows CE devices can be found by first look in the registry in `HKLM\Drivers\Active`. A group of registry keys with numeric names will be found at that location. Looking at the "Name" string inside of each one of those keys will give the name of an item of hardware. In the sample program `CePowerState` I enumerate the hardware with the following code:

```
// Get the names of all of the subkeys that  
// refer to hardware on the device.  
RegistryKey driverKeyRoot =  
Microsoft.Win32.Registry.LocalMachine.OpenSubKey("Drivers\\Active");  
string[] keyName = driverKeyRoot.GetSubKeyNames();  
  
//We are saving this information to list for sorting later  
List<string> deviceNameList = new List<string>();  
for (int i = 0; i < keyName.Length; ++i)  
{  
    //Get the name of the hardware and add it to the list  
    RegistryKey currentKey = driverKeyRoot.OpenSubKey(keyName[i]);  
    string deviceName = currentKey.GetValue("Name") as string;  
    if(deviceName!=null)  
        deviceNameList.Add(deviceName);  
}  
//Sort the list  
deviceNameList.Sort();  
//Add the list to the list box so the user can select hardware  
for (int i = 0; i < deviceNameList.Count; ++i)  
{  
    lstDeviceList.Items.Add(deviceNameList[i]);  
}
```

The sample program **CePowerState** will enumerate all of the hardware found in this location and allow you to directly set the power state of the hardware by calling

SetDevicePower. The screen backlight is usually (but not always) named BKL1: and the sound device is usually name WAV1:. Setting the power state of the backlight to D0 will turn it on and D4 will turn it off. If you are playing audio and set the power state of WAV1 to D4 then the sound will no longer be heard until the power state is set back to D0. This is the code that sets the device's power state.

```
//Get the name of the selected hardware
string deviceName = lstDeviceList.SelectedItem as string;
//Get the power state to which the device will be changed
CEDEVICE_POWER_STATE state =
(CEDEVICE_POWER_STATE)Enum.Parse(typeof(CEDEVICE_POWER_STATE),
lstPowerState.SelectedItem as string,true);
//deviceHandle = CoreDLL.SetPowerRequirement(deviceName, state, (DevicePowerFlags)1 ,
IntPtr.Zero, 0);
CoreDLL.SetDevicePower(deviceName, DevicePowerFlags.POWER_NAME, state);
```

## Registered Device Power Settings for Power States

There are default device power settings for certain power states. For example, in unattended state my Windows Mobile Professional phone turns off the backlight and the screen but keeps the sound adapter and the GPS receiver powered on. The information on these settings can be found in

**HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Power\Timeouts\State\**. For each one of the power states you will find a key. The contents of this key are string sub keys whose names match hardware on the device and whose value math a device power state (so 0x04 matches D4 which means a certain piece of hardware will be on at full power in the matching state).

## Preventing the System from Powering Down

Professional devices will automatically progress towards their suspended states if no users actions are detected. There are times at which you may need for a device to maintain its full power state even though the user isn't directly interacting with the system (such as when listening to music with Windows Media Player). To prevent the system from powering down from being idle periodically make calls to SystemIdleTimerReset(). Calling this function prevents the suspend timeout from being reached. The suspend timeouts can be changed on a system so you will need to query for the suspend values and ensure that you call SystemIdleTimerReset on an interval shorter than the smallest suspend timeout value. The suspend timeout values can be acquired by calling SystemParametersInfo. This function can be used to acquire the following three timeout values.

Timeout Type	Description
<b>SPI_GETBATTERYIDLETIMEOUT</b>	Timeout from the last user input until suspend when running off of battery power.
<b>SPI_GETEXTERNALIDLETIMEOUT</b>	Time out from the last user input until suspend when running on external power
<b>SPI_GETWAKEUPIDLETIMEOUT</b>	The timeout from the system powering itself on before it suspends again

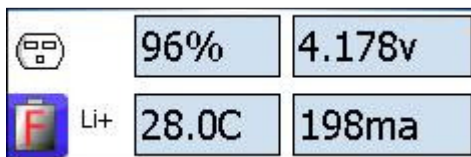
The timeout values can also be retrieved from the registry. Look in location **[HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Power]** for the following keys.

Key	Description
<b>BatteryPowerOff</b>	The amount of time (When running on battery) that the system will be idle before it suspends. Set to zero to disable suspending.
<b>DisableGwesPowerOff</b>	Setting to a non-zero value disables GWES's management of system suspending
<b>ExtPowerOff</b>	Amount of time before system suspends when running on external power. Set to zero to disable suspending.
<b>ScreenPowerOff</b>	The amount of time that the system will be idle before the screen powers off
<b>WakeupPowerOff</b>	The amount of time the system will wait on user input before suspending after waking up due to a non-user wakeup event.

## The Battery State

The state of the batteries in Windows Mobile devices is available through the native method [GetSystemPowerStatusEx2\(\)](#). The information provided will tell you whether or not the device is connected to an external power source and an estimate of the amount of power left in the primary and backup battery (if present). The native function returns its information in a [SYSTEM\\_POWER\\_STATUS\\_EX2](#) structure and contains information not available through the managed SystemState class such as the temperature of the battery or the instantaneous amount of power being consumed by the battery. The accuracy of the battery metrics returned is dependent on the OEM. The OEM also may have decided not to gather and return certain battery metrics in which case one of the xxxx\_UNKNOWN values could be returned. The maximum amount of time between the OS updating its information on the battery can be found in **[HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Power\Timeouts\BatteryPoll]**.

In the attached sample programs there is a program called "Battery Status" that displays the information on the battery. The top of the program displays the most important data on the battery; its current capacity, whether or not it is connected to an AC source, the battery's temperature, the voltage output, and the battery's current (note that the sign on the current will change when the device is charging).



## Preserving Power

If you would like to ensure that your programs act responsibly and don't unnecessarily burn through the battery's power then the best thing that your program can do is to stop

performing work when no work needs to be performed. Sounds like common sense, doesn't it. But you would be surprised of the unnecessary amount of work that programs do when they could save power by doing nothing. The most common power burning pattern I come across in the support forums is continuous polling. An example of such an implementation follows.

```
BeginSomeTimeConsumingTask();  
while(!taskComplete)  
{  
    Application.DoEvents();  
}
```

Most Windows applications spend a majority of their life time waiting on something to occur. That something could be the user pressing the next key, a file opening, or waiting on a time out. When an even occurs the program will respond to it and then go back to waiting. A system can take advantage of the waiting by running the processor in a lower power state. But the above code isn't waiting. It is spinning on a variable and utilizing as much CPU bandwidth as it can consume, it is unnecessarily asserting cycles across the devices memory bus to read the variable and burning away at the devices batter at a much faster rate than normal. The better implementation is to have the working thread signal the completion of its task through either a system event or a .Net event. For programs that run on Windows Mobile Standard devices when the program is not visible then do not refresh the programs display. Continuing to update the display will result in significantly lower battery life.

## Power Scenarios and Solutions

The following are common scenarios that involve power management and their solutions.

## Problem – Iterative Checking

### Description

A thread needs to wait for an event to occur before moving on to perform the rest of its actions. Many developers will create a Boolean variable and will change its value when the other thread can continue. The waiting thread polls this variable until it changes, then it continues its task.

### Explanation

The waiting thread is unnecessarily polling a variable and burning through CPU cycles while not accomplishing any work.

### Solution

There are already facilities to implement this same functionality provided by the runtime and the operating system through various synchronization objects. Synchronization techniques could be used instead to block a thread until a task is complete or another thread signals for it to complete. A program can also wait on an external event or register itself to be started during a certain external event (such as the device being connected to power or an ActiveSync session being initialized). For these situations the program can register to be notified or started up when the event occurs. When polling is the only way to query for status then consider putting your thread to sleep between checks.

### Example : No Polling

For a simple example I will use delegates and the thread pool to perform my long running tasks. The delegates are necessary since UI elements cannot be updated from a secondary thread and delegates can be used to ensure UI updates occur on the correct thread. The complete program can be found in the attached sample code.

```
delegate void UpdateStatusDelegate(int progress);
delegate void VoidDelegate();

UpdateStatusDelegate _updateStatus;
VoidDelegate _taskComplete;
WaitCallback _longRunningTask;

public Form1()
{
    InitializeComponent();
    _longRunningTask = new WaitCallback(LongRunningTask);
    _updateStatus = new UpdateStatusDelegate(UpdateStatus);
    _taskComplete = new VoidDelegate(TaskComplete);
}

// We cannot update UI elements from secondary threads. However
// the Control.Invoke method can be used to execute a delegate
// on the main thread. If UpdateStatus is called from a secondary
// thread it will automatically call itself through Control.Invoke
```

```

// to ensure its work is performed on the primary (UI) thread.
void UpdateStatus(int progress)
{
    if (this.InvokeRequired)
        BeginInvoke(_updateStatus, new object[] { progress });
    else
    {
        this.txtFeedback.Text = progress.ToString();
        pbWorkStatus.Value = progress;
    }
}

// The long running task is contained within this method.
// as the task runs it will pass progress updates back to
// the UI through the UpdateStatus method. Upon completion
// of the task a call is made to TaskComplete
void LongRunningTask(object o)
{
    try
    {
        for (int i = 0; i < 100; ++i)
        {
            Thread.Sleep(100);
            UpdateStatus(i);
        }
        TaskComplete();
    }
    catch (ThreadAbortException exc)
    {
        //The task is being cancelled.
    }
}

// Since the actions the program takes at completion of the long
// running task touch UI elements the TaskComplete method will
// also call itself (if necessary) through Control.Invoke to
// ensure that it is executing on the primary (UI) thread.
void TaskComplete()
{
    if (this.InvokeRequired)
    {
        this.Invoke(_taskComplete);
    }
    else
    {
        pbWorkStatus.Value = 0;
        miWork.Enabled = true;
        txtFeedback.Text = "Complete";
    }
}

// When the user selects the menu item to begin working
// I will disable the work menu item to prevent concurrent
// request and start the long running progress on a secondary
// thread.
private void miWork_Click(object sender, EventArgs e)
{
    miWork.Enabled = false;
    ThreadPool.QueueUserWorkItem(_longRunningTask);
}

```

```
private void miQuit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

## Problem – Don't Let the Device Sleep

### Description

A program is performing a long running task that does not require user interaction such as a GPS navigation (for which the user would probably look at the screen but not touch it). After the program has run for an amount of time the screen turns off and disrupts the user's experience.

### Explanation

Since the user is not interacting with the system it does as it normally should and is powering the device down to save power.

### Solution

Call the native function `SystemIdleTimerReset()` periodically to prevent the device from powering down.

### Example Program : PreventSleep

The program "PreventSleep" reads the idle timeout values from the registry and will call `SystemIdleTimeReset` at an interval that is slightly shorter than the shortest timeout. As a result for as long as `PreventSleep` is running the device will not go to sleep due to lack of user interaction.

In the code example `SystemIdleTimeReset` is called on a timer so that the main thread does not need to concern itself with maintaining the resets.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using Win32;
using Microsoft.Win32;

namespace PreventSleep
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            // Look in the registry to see what the shortest timeout
            // period is. Note that Zero is a special value with respect
```

```

// to timeouts. It indicates that a timeout will not occur.
// As long as SystemIdleTimeerReset is called on intervals
// that are shorter than the smallest non-zero timeout value
// then the device will not sleep from idleness. This does
// not prevent the device from sleeping due to the power
// button being pressed.
int ShortestTimeoutInterval()
{
    int retVal = 1000;
    RegistryKey key =
Registry.LocalMachine.OpenSubKey(@"\SYSTEM\CurrentControlSet\Control\Power");
    object oBatteryTimeout = key.GetValue("BattPowerOff");
    object oACTimeOut = key.GetValue("ExtPowerOff");
    object oScreenPowerOff = key.GetValue("ScreenPowerOff");

    if (oBatteryTimeout is int)
    {
        int v = (int)oBatteryTimeout;
        if(v>0)
            retVal = Math.Min(retVal,v);
    }
    if (oACTimeOut is int)
    {
        int v = (int)oACTimeOut;
        if(v>0)
            retVal = Math.Min(retVal, v);
    }
    if (oScreenPowerOff is int)
    {
        int v = (int)oScreenPowerOff;
        if(v>0)
            retVal = Math.Min(retVal, v);
    }

    return retVal*9/10;
}

private void Form1_Load(object sender, EventArgs e)
{
    // Set the interval on our timer and start the
    // timer. It will run for the duration of the
    // program
    int interval = ShortestTimeoutInterval();
    resetTimer.Interval = interval;
    resetTimer.Enabled = true;
}

private void miQuit_Click(object sender, EventArgs e)
{
    this.Close();
}

// Call the SystemIdleTimerReset method to prevent the
// device from sleeping
private void resetTimer_Tick(object sender, EventArgs e)
{
    CoreDLL.SystemIdleTimerReset();
}
}
}

```

## Problem – Wake up and Quietly Work

### Description

Your program needs to wake up and perform some work at a predetermined time

### Explanation

When the device wakes up and your program begins to work placing the device in a full powered mode will unnecessarily get the attention of the user or may lead to screens and buttons receiving key presses if the device were in the users problem. We need to wake up the device without powering up the screen. The user should be totally unaware that the device is doing any work.

### Solution

When the device wakes up the program should promptly place the device Windows Mobile device in unattended mode. In this mode if the user were not using the device then the device's screen is not going to power up, yet the program is able to run as though the device were fully powered. Once the program completes its work it can release its requirement for the device to be in unattended mode and can go back to sleep with user never being made aware that the device had done anything.

### Example Program : Work Quietly

WorkQuietly allows the user to select a sound file and a time delay when started. After selecting the "Run" menu option the program will schedule for a restart of itself using the CeRunAppAtTime native call as described in a [previous article](#) on scheduled launches of programs and terminate. After the selected time delay the program will start and play a sound. Playing a sound is a complete contradiction of the programs name, but it is the most noticeable thing to do to prove to someone that the program is working.

This is one of the rare cases in which I modify the Main() method of a project. When an application is started because of scheduling the command line argument "AppRunAtTime" is passed. The default implementation of the Main() method doesn't accept command line arguments. When the AppRunAtTime argument is found instead of loading the form I have the application play its sound and quit.

```
// Schedule the execution of the program and terminate.
// You can either put the device to sleep or leave it at
// full power. In either case the program will run at its
// assigned time, play a sound, and then terminate
private void miRun_Click(object sender, EventArgs e)
{
    int waitTime = int.Parse(this.cboStartTime.Text);
    DateTime startTime = DateTime.Now.AddSeconds(waitTime);
    string targetExecutable =
this.GetType().Assembly.GetModules()[0].FullyQualifiedName;
    RunAppAtTime(targetExecutable, startTime);
    using (StreamWriter sw = new StreamWriter(targetExecutable +
".soundPath"))
    {
        sw.Write(txtSoundPath.Text);
    }
}
```

```

        sw.Close();
    }
    this.Close();
}

```

The following is my modified main method.

```

[MTAThread]
static void Main(string[] args)
{
    if (args.Length == 0)
        Application.Run(new Form1());
    else if (args[0].Equals("AppRunAtTime"))
    {
        string soundPath;

        // We started due to a scheduled event
        CoreDLL.PowerPolicyNotify(PPNMessage.PPN_UNATTENDEDMODE, -1);
        string targetExecutable =
typeof(Form1).Assembly.GetModules()[0].FullyQualifiedName;

        StreamWriter argInfo = new StreamWriter(targetExecutable +
".argument.txt");
        argInfo.WriteLine(args[0]);
        argInfo.Close();

        using (StreamReader sr = new StreamReader(targetExecutable +
".soundPath"))
        {
            soundPath = sr.ReadToEnd();
            sr.Close();
        }
        if (File.Exists(soundPath))
            Aygshell.SndPlaySync(soundPath, 0);
        CoreDLL.PowerPolicyNotify(PPNMessage.PPN_UNATTENDEDMODE, 0);
    }
}

```

## Problem – Your program needs to Query the Power level of the Battery

Being able to query the power level of the battery is useful for creating Windows Mobile utilities and for making programs that respond responsibly to available power levels.

### Example Program : BatteryStatus

The process of acquiring the battery status information is incredibly simple. The P/Invoke declaration and a helper function are displayed below.

```

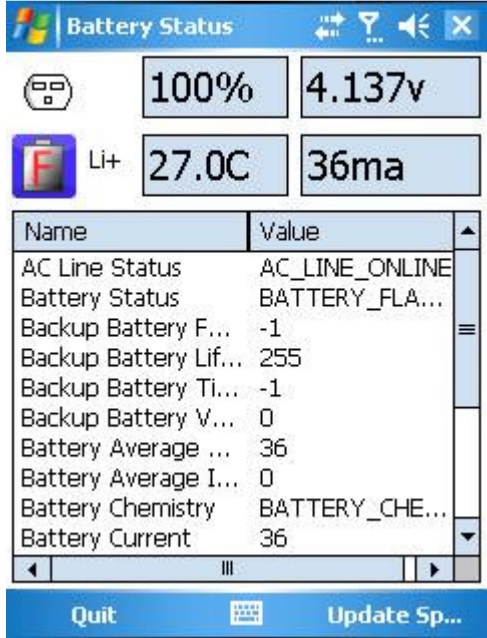
[DllImport("CoreDLL")]
public static extern int GetSystemPowerStatusEx2(
    SYSTEM_POWER_STATUS_EX2 statusInfo,
    int length,
    int getLatest
);

public static SYSTEM_POWER_STATUS_EX2 GetSystemPowerStatus()
{

```

```
SYSTEM_POWER_STATUS_EX2 retVal = new SYSTEM_POWER_STATUS_EX2();
int result = GetSystemPowerStatusEx2( retVal, Marshal.SizeOf(retVal) , 0);
return retVal;
}
```

The parameter `getLatest` controls whether or not cached information is returned or more updated information is returned. The cached information is typically no older than 5 seconds, there is usually no need to demand more recent information. If you would like to ensure that your information is always up to date then pass a non-zero value for the `getLatest` parameter. The sample program displays the most interesting information up top with icons and displays all of the values returned by the function in a list box below the graphics. The information at the top includes the battery charge left, the voltage output of the battery, the battery temperature, the current being consumed, and whether or not the battery is running off of AC power. Note that the sign on the current may change between positive and negative when the battery goes from charging to discharging.



### Problem – Maintain Power to Hardware while in Sleep Mode

On some devices when the device enters sleep mode (the mode where the screen is off but the processor is still running) hardware that is needed by a program may have its power interrupted and is unavailable until the device wakes up.

#### Explanation

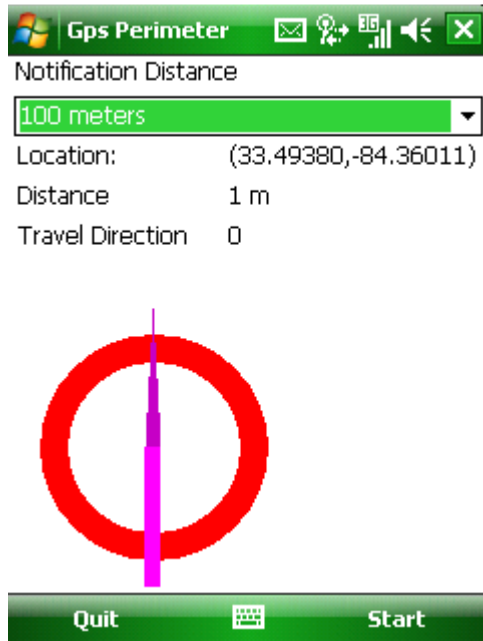
The device’s power profile may be set to power down hardware that the OEM felt was unnecessary when the device is in sleep mode. The screen and the screen’s backlight will invariably be amongst hardware to be powered down. Occasionally the GPS device or the Bluetooth radio may have been set to power down on some devices in these modes. When the device sleeps the power is cut off to these accessories and a program running in unattended mode finds itself unable to access hardware that it had access to before.

## Solution

A program can ensure that it has access to the hardware that it needs in sleep mode by setting a power requirement for the hardware. Regardless of the default power state for the hardware in sleep mode the program's requirements will be satisfied. While GPS devices are not supported by the .Net framework the Windows Mobile 5 Professional SDK includes an implementation of a GPS class that makes the appropriate low level calls to interact with the GPS intermediate driver.

## Example Program: GPS Perimeter

**For this example your device must have a GPS receiver**



The example program will alert a user when he or she travels a distance from a set point. Once the user travels outside of the area the device will play sounds and display a message on the interface alerting the user that he or she is outside of the set perimeter. If the device had gone to sleep from the time that the user started it the program will still be able to keep track of the user's position and alert the user through sound.

When the GPS Perimeter program starts it first makes known that it runs in unattended mode. By doing this if the device sleeps it will not enter the suspended state. The program then sets the power requirement for the GPS receiver to full power.

```
public Form1()
{
    InitializeComponent();
    CoreDLL.PowerPolicyNotify(PPNMessage.PPN_UNATTENDEDMODE, -1);
}

private void Form1_Load(object sender, EventArgs e)
{
    _gpsPowerRequirements = CoreDLL.SetPowerRequirement(
```

```

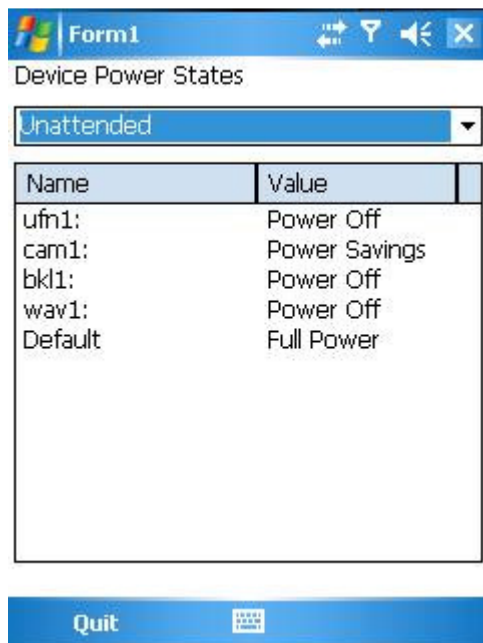
        GpsDeviceName,
        CEDEVICE_POWER_STATE.D0,
        DevicePowerFlags.POWER_NAME,
        IntPtr.Zero,
        0);

    _gpsDevice = new Gps();
    _gpsDevice.Open();
    _gpsDevice.LocationChanged += new
LocationChangedEventHandler(_gpsDevice_LocationChanged);
    _gpsDevice.DeviceStateChanged += new
DeviceStateChangedEventHandler(_gpsDevice_DeviceStateChanged);
}
private void Form1_Closing(object sender, CancelEventArgs e)
{
    _gpsDevice.Close();
    CoreDLL.PowerPolicyNotify(PPNMessage.PPN_UNATTENDEDMODE, 0);
    CoreDLL.ReleasePowerRequirement(_gpsPowerRequirements);
}

```

## Curiosity: What power states are supported on my device?

I stumbled across some interesting registry keys while searching for other information which resulted in this program. Using the registry this program will list your device's power states. Selecting one of the power states will cause the program to inform you of the default power state for the devices driver's along with the hardware that should have some state other than that default.



## Example Program : MyPowerStates

The code for this does nothing more than a filtered registry dump. Here's the source code of the main part of the program

```

        const string BASE_POWER_HIVE =
@"System\CurrentControlSet\Control\Power\State";
        string[] _powerStateNames = {"Full Power", "Power Savings", "Standby", "Sleep
Mode", "Power Off"};

        Regex _targetRegistryValue = new Regex("(DEFAULT)|(^.*:$)",
RegexOptions.IgnoreCase);

        string[] GetPowerStateList()
        {
            RegistryKey powerStateKey =
Registry.LocalMachine.OpenSubKey(BASE_POWER_HIVE);
            return powerStateKey.GetSubKeyNames();
        }

        string[][] GetPowerStateInfo(string stateName)
        {
            RegistryKey stateInformationKey =
Registry.LocalMachine.OpenSubKey(String.Format(@"{0}\{1}", BASE_POWER_HIVE,
stateName));
            string[] valueList = stateInformationKey.GetValueNames();
            List<StateInfo> = new List<StateInfo>();
            for (int i = 0; i < valueList.Length; ++i)
            {
                string currentValue = valueList[i];
                if (_targetRegistryValue.IsMatch(currentValue))
                {
                    StateInfo.Add(new string[] { valueList[i], _powerStateNames[(int)
stateInformationKey.GetValue(currentValue)]});
                }
            }
            return StateInfo.ToArray();
        }

        void PopulatePowerState()
        {
            cboPowerState.Items.Clear();
            string[] stateList = GetPowerStateList();
            List<string> sortList = new List<string>(stateList);
            sortList.Sort();

            for (int i = 0; i < sortList.Count; ++i)
            {
                cboPowerState.Items.Add(sortList[i]);
            }
        }

```

## Resources and References

### API References

Power Management Functions <http://msdn.microsoft.com/en-us/library/aa909892.aspx>

### Articles and Blog Entries

Name	Author	URL
------	--------	-----

Automatically Start your Program on Windows Mobile	Joel Ivory Johnson	<a href="http://tinyurl.com/62jqoy">http://tinyurl.com/62jqoy</a>
Power to the System	Mike Calligaro	<a href="http://tinyurl.com/6gu4bb">http://tinyurl.com/6gu4bb</a>
How Windows CE Bus Drivers Work	David Liao	<a href="http://tinyurl.com/5sqgig">http://tinyurl.com/5sqgig</a>
Keeping your UI Responsive and the Dangers of Application.DoEvents		<a href="http://tinyurl.com/5m2uns">http://tinyurl.com/5m2uns</a>

## Books

Programming Windows Embedded CE 6.0, Douglas Boling , Microsoft Press Copyright 2008

Mobile Development Handbook, Andy Wigley, David Moth, Peter Foot, Microsoft Press Copyright

## History

- 28 August 2008 - Initial Publication
- 2 September – Addition of GPS Perimeter project