# Writing Energy-Efficient Windows* Store Applications for Mobile Devices

## Impact of CPU Utilization on Platform Power

### Outlines

1. Introduction
2. Processor Power States
   1.1 Active state
   1.2 Sleep state
   1.3 Power state management
   1.4 Experimental setup
3. Timer Impact
   3.1 Create/cancel a thread pool timer
   3.2 Periodic timer interval impact
   3.3 P and C state residency
4. ThreadPoolTimer versus DispatcherTimer
   4.1 Create/cancel a dispatcher timer
   4.2 UI thread frame rate and power comparison
5. Summary
6. Future discussion
7. About the Author
8. Notices
1. Introduction

Mobile devices are everywhere today, like smartphones, tablets, Ultrabook™ devices, etc. All of them are powered by batteries. Battery life time is critical to end users because everyone wants the battery to last as longer as possible.

The power drain has direct impact on the battery life time. It is affected by the whole mobile app stack. As an important layer in the software stack, mobile apps play a critically important role on power consumption of the platform. End users interact with apps directly and the apps directly affect the power profile of platforms. For app developers, one significant task is to write energy-efficient apps to reduce power consumption such that the device battery life time is extended.

Different apps can trigger the power usage of different hardware components like CPU, graphical component, storage media, network, sensors, etc. In the series of writing energy-efficient apps, we are going to discuss energy-efficient design guidelines for different hardware components. The guidelines are based on a series of best practices for case studies with the Windows Store apps, implemented by functions/classes in the new API set – Windows Runtime (aka WinRT).

In this article, we focus on the CPU power impact. We first address the processor power states and then discuss how CPU power is impacted by some WinRT API usages.

2. Processor Power States

Modern processors have multiple power states. These power states can be classified into two categories: active states and sleep states. The active state includes multiple performance states (aka P states) depending on the supply voltage/frequency. The sleep states include multiple C states (aka Cx states) depending on the power savings. Please refer to the specific IA-based processor specification document for the detailed definitions of active states and sleep states.

2.1 Active State

Active state (aka C0 state) means the processor is executing instructions. Active state includes multiple P states with different supply voltages / frequencies. A particular P state is associated with a specific supply voltage and frequency. Generally a higher P state is associated with a higher supply voltage and a higher frequency. The dynamic power consumption at a particular P state can be captured by this equation:

\[ P = C_{\text{dyn}} V^2 F \]

Here \( P \) is the dynamic power consumption of a particular P state. \( C_{\text{dyn}} \) represents the dynamic workload on CPU. \( V \) and \( F \) represent the supply voltage and frequency in the P state. The equation implies that the CPU consumes more dynamic power at a higher P state.
2.2 Sleep State
When the processor is idle, it enters the sleep state to save power. Many IA-based processors include multiple sleep states (aka Cx states) like C1, C2/C3, C5, C6, and C7. C1 state typically halts the processor due to no instruction execution. C2/C3 state, if present, offers improved power savings over C1 state. In C2/C3 state the processor’s cache remains on with major function units off. C5/C6/C7 state, if present, is the additional deep sleep states. From C1 state to C7 state, the processor power is reduced with longer wakeup latency.

2.3 Power State Management
The power states are managed by the processor driver and the operating system. To improve the power savings, a typical scheduling mechanism is widely performed. (Note that the scheduling mechanism on current operating systems and processor drivers is more complex than the described mechanism.) When the processor is active with low utilization, the processor is scheduled to enter a lower P state to save power in active state. When the processor idle residency time is long, the processor enters a deeper sleep state to save power. App developers can improve the performance of their applications to reduce the CPU utilization and use a less frequent timer to reduce CPU power consumption.

2.4 Experimental Setup
We have case studies on a typical timer usage in Windows Store app. The experiments were executed on an UltraBook with the 3rd generation Intel® Core™ processor and a clean Windows 8 RTM build. We utilized the Intel® Power Gadget to collect the processor package power. The display refresh rate of the platform is 60 frames per second (fps).

3. Timer Impact
Generally, higher frequency timers in applications can trigger more processor wakeups and thus lead to higher CPU utilizations. A periodic timer may have a big impact on the processor power. Here is a case study on the CPU power impact of a periodic timer implemented in a Windows Store app.

3.1 Create/cancel a Thread Pool Timer
Windows Runtime provides a ThreadPoolTimer class. The software timers in apps can be created by calling the CreateTimer or CreatePeriodicTimer functions in the class. The timers are widely utilized for thread creation in Windows Store apps. We implemented the periodic timer by calling CreatePeriodicTimer function.
The sample code shows how to use the API to create a periodic timer.

```csharp
private void CreatePeriodicTimer(object sender, RoutedEventArgs args)
{
    if (int.TryParse(PeriodMs.SelectionBoxItem.ToString(),
                    out ThreadPoolSample.PeriodicTimerMilliseconds))
    {
        ThreadPoolSample.PeriodicTimer =
            ThreadPoolTimer.CreatePeriodicTimer((timer) =>
            {
                System.Threading.Interlocked.Increment(ref
                    ThreadPoolSample.PeriodicTimerCount);
                Dispatcher.RunAsync(CoreDispatcherPriority.High, () =>
                {
                    ThreadPoolSample.PeriodicTimerScenario.
                        UpdateUI(Status.Completed);
                        
                });
            }, TimeSpan.FromMilliseconds(
                ThreadPoolSample.PeriodicTimerMilliseconds));
            //Do nothing or update XAML textbox
            UpdateUI(Status.Started);
    }
}
```

Sample Code 1. Create a periodic timer with `ThreadPoolTimer` class

The sample code shows how to use the API to cancel a periodic timer.

```csharp
private void CancelPeriodicTimer(object sender, RoutedEventArgs args)
{
    if (ThreadPoolSample.PeriodicTimer != null)
    {
        ThreadPoolSample.PeriodicTimer.Cancel();
        ThreadPoolSample.PeriodicTimerCount = 0;
        UpdateUI(Status.Canceled);
    }
}
```

Sample Code 2. Cancel a periodic timer with `ThreadPoolTimer` class

### 3.2 Periodic Timer Interval Impact

We compare the periodic timers with various interval settings fired without workload and with a simple XAML textbox UI update. Figure 1 plots the average processor package power consumption relative to the average processor idle power for various experiments.
When the periodic timers are fired without workload, the processor power consumption is close to the idle power even though the timer frequencies are different. This implies that the processor driver and operating system optimize the timer scheduling and maintain a low processor power for timers with light workload. When the periodic timers are fired with a simple UI update, as the timer interval setting decreases the processor power increases exponentially up to 1.72 times the processor idle power. From the power consumption trend, the app developer needs to know what is triggered by the periodic timer in his/her app. We recommend app developers choose a suitable timer interval to satisfy the app requirement with acceptable processor power overhead. Once the timer is not needed, the timer should be cancelled.

3.3 P and C State Residency

P state residency and C state residency indicate the time percentage when a processor stays in a particular P or C state during a time period. Generally, the summation of all the C state residencies (C0-C7) is equal to 100% and the summation of all the P state residencies (all P states) is equal to 100%. For the case study on a periodic timer fired with a simple XAML update, the P and C state residency information is summarized in table 1.
From Table 1, when the timer interval is set to a small value (5 or 16 milliseconds), the C7 residency values are all zeros. This is because the processor doesn’t enter the deepest sleep state because the timers are fired too frequently. The processor has to wake up from C6 state to update the UI. When the timer intervals are extended (20 or 100 milliseconds), the processor spends around 70% of the time in C7 state. Therefore, the processor’s power is reduced in sleep state.

The P state residency shows when the timer interval is set to small values the processor spends 100% of active time at a rated frequency. Note that the rated frequency is not the lowest P state. When the timer interval settings are extended, the processor has a chance to degrade to a lower P state. Therefore, the processor’s power is reduced in active state.

This means the software timer interval settings associated with the workload have a large impact on the P and C state residencies. We recommend app developers monitor the P and C state residencies to reduce the processor power caused by the application when the direct processor power measurement is not available.

### 4. ThreadPoolTimer versus DispatcherTimer

Windows Runtime provides another timer in the `Windows.UI.Xaml.DispatcherTimer` class. We compare the two timers for a simple XAML UI update. The `DispatcherTimer` provides a timer that is integrated into the `Dispatcher` queue, which is processed at a specified interval of time and at a specified priority.
4.1 Create/cancel a Dispatcher Timer

The *DispatcherTimer* is created as shown in the sample code below.

```csharp
private void CreateDispatcherTimer(object sender, RoutedEventArgs args)
{
    if (int.TryParse(PeriodMs.SelectionBoxItem.ToString(),
                     out ThreadPoolSample.PeriodicTimerMilliseconds))
    {
        dispatcherTimer = new DispatcherTimer();
        dispatcherTimer.Tick += UpdateUIwithDispatcherTimer;
        dispatcherTimer.Interval = TimeSpan.FromMilliseconds(
                                   ThreadPoolSample.PeriodicTimerMilliseconds);
        dispatcherTimer.Start();
    }
}
```

*Sample Code 3. Create a periodic timer with DispatcherTimer class***

```csharp
private void CancelDispatcherTimer(object sender, RoutedEventArgs args)
{
    dispatcherTimer.Stop();
}
```

*Sample Code 4. Cancel a timer with DispatcherTimer class***

4.2 UI Thread Frame Rate and Power Comparison

We implement the two timers in a Windows Store app to update a simple XAML textbox with various periodic timer interval settings. The UI thread frame rate and the processor power are compared in Figure 2. The lines represent the average processor package power consumption relative to the processor idle power. The bars represent the UI thread frame rate for the XAML textbox update.
The figure shows at a particular periodic timer interval setting the processor consumes less power for the `DispatcherTimer` implementation. This is because the `DispatcherTimer` class limits the UI thread frame rate and has a maximum update frame rate (30 fps) at half of the display refresh rate (60 fps). Therefore, we recommend app developers utilize `DispatcherTimer` class for UI updates.

5. Summary

In the article we explain the available processor power states. Then we discuss how to save CPU power in a Windows Store app. We expect developers can apply this guideline in the software development cycle.

6. Future Discussion

As mentioned in the introduction session, we will discuss the power impact of other hardware components in future articles in this series. We will address how to save power in other components for various types of Windows Store apps like graphical intensive apps, file-access based apps, network I/O-based apps, sensor usages, etc.
7. About the Author

Sushu Zhang is a software engineer in Intel’s Software and Services Group. She is working on ISV scale enabling for Intel-based platforms with Android* / Windows OS. She developed the Intel’ Power Monitoring Tool for Android Devices. Most recently, Sushu has been involved with several energy-efficiency projects at Intel. Prior to joining Intel, Sushu worked at Microsoft on Windows energy efficiency. Sushu earned a Ph.D in Computer Science at the Arizona State University. Her research area was system-level power and thermal management.

8. Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to:  http://www.intel.com/design/literature.htm

Intel, the Intel logo Core, and Ultrabook are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others
**This sample source code is released under the Intel Sample Source Code License Agreement**
Copyright© 2012 Intel Corporation. All rights reserved.