OPTIMIZING WINDOWS* 8 APPLICATIONS FOR CONNECTED STANDBY

Whitepaper

Manuj Sabharwal (Software and Services Group)
Abstract
This white paper describes how to validate and analyze the behavior of Windows 8* applications during connected standby which is one of the Microsoft* WHQL requirement for Windows 8 [1]. It explains how to identify applications that drain the battery excessively during connected standby and the steps necessary to mitigate that problem. This document is intended for software developers, Original Equipment Manufacturers, and technical consumers.

Introduction
The connected standby feature enables the system to stay up-to-date and reachable whenever network connectivity is available. Much like how a phone maintains connectivity to the cellular network while the screen is off, Windows 8 applications written for connected standby are able to deliver an up-to-date experience immediately after returning from a low power state. More information on connected standby on PCs can be obtained from Microsoft [1].

When the display is turned off on connected standby capable systems, all running software (including applications and operating system software) become subject to a new set of activity restrictions. Windows Desktop Activity Moderator (DAM) suppresses legacy app execution in a manner similar to the Sleep state. It does this by suspending all applications in the user session and throttling all 3rd party services to create predictable power usage over the period of idle time. This enables systems that support connected standby to deliver minimized resource usage and long, consistent battery life while enabling Windows 8 Modern UI apps to deliver the connected experiences they promise. In addition, as hardware power states become more sensitive, software services must be well behaved at connected standby so they don’t needlessly wake/throttle the system, which would limit battery life.

The rest of this paper details tools and techniques for understanding system behavior during connected standby, and then presents two case studies of applications that can improve their behavior during connected standby.

Tools

We used two easily available developer tools to understand the behavior of applications running in connected standby, as described in this section.

Windows PowerCfg
Windows PowerCfg is a command-line utility used to control power settings. It uses Event Tracing for Windows (ETW) to profile systems. Users can use Windows PowerCfg to view and
modify power plans and settings such as standby time, wake timer, and power schemes. Running Powercfg with the “-energy” option analyzes common energy-efficiency and battery life problems, such as platform timer-tick setting changes, changes to timers by application processes or DLLs, and processor utilization per process. With this setting, it will also validate if the system supports connected standby and will report power-management settings in hardware and in the OS. Administrator permission is required to run Windows Powercfg.

Two command line options are used to validate and find information on behavior during connected standby:

**Powercfg –a:** This option reports the available sleep states of the system. To try it, open a command window from Windows. At the prompt type: %powercfg –a

A system that supports connected standby will report the supported sleep states available on the system and list Connected Standby as a supported state. The command will report that this system will not support “Hybrid Sleep” or S3 state. Figure 1 shows output of powercfg –a on connected standby system.

![Powercfg -a output](image)

**Figure 1: Powercfg -a output**

**Powercfg -batteryreport**

The “-batteryreport” option provides information on connected standby support and other related information. It creates an HTML report of the system battery life statistics by collecting a profile based on always-running, built-in system tracing. The report provides a summary of the battery installed, BIOS version, connected standby support, recent usage, and a battery life estimate based on actual use of the system including connected standby usage. Figure 2 shows sample output of “-batteryreport” when run on a PC that supports connected standby.
The report also provides battery usage when the system was in active, suspended and connected standby states, as shown in Figure 3.

More information on the use of Windows Powercfg can be found at this [Microsoft website] [2].
Microsoft Windows Performance Analyzer

Windows Performance Analyzer (WPA), also known as xperf, is a set of performance monitoring tools used to produce in-depth performance and power profiles of Microsoft Windows and of applications. WPA is useful to troubleshoot power hygiene problems.

Before we go into the case study, you should understand the terminology of WPA. These are the definitions of key terms and column names in WPA, taken from the System Internals documentation at [3]:

- **Ready Thread**: Thread in the ready state is waiting to execute or ready to be in-swapped after completing a wait. When looking for a thread to execute, the dispatcher considers only the pool of threads in the ready state.
- **Standby**: A thread in the standby state has been selected to run next on a particular processor. When the correct conditions exist, the dispatcher performs a context switch to this thread. Only one thread can be in the standby state for each processor on the system. Note that a thread can be preempted out of the standby state before it ever executes (if, for example, a higher priority thread becomes runnable before the standby thread begins execution).
- **Waiting**: A thread can enter the waiting state in several ways: a thread can voluntarily wait for an object to synchronize its execution, the operating system can wait on the thread’s behalf (such as to resolve a paging I/O), or an environment subsystem can direct the thread to suspend itself. When the thread’s wait ends, depending on the priority, the thread either begins running immediately or is moved back to the ready state.
- **CPU Precise**: The CPU Usage (Precise) graph records information associated with context switch events. Each row represents a collection of data associated with a single context switch, when a thread started running.
- **% CPU Usage**: The CPU usage of the new thread after it is switched in expressed as a percentage of total CPU time over the currently visible time range.
- **Count**: The number of context switches represented by the row (always 1 for individual rows).
- **NewThreadId**: The thread ID of the new thread.
- **NewThreadStack**: The stack of the new thread when it is switched in.
- **ReadyingProcess**: The process owning the readying thread.
- **SwitchInTime**: The time when the new thread was switched in.
- **LastSwitchOutTime**: The time when the new thread was last switched out.
- **TimeSinceLast**: SwitchInTime - LastSwitchOutTime
Figure 4 shows the key column names in WPA UI.

**Generic Events:** User provided events are populated to analyze kernel trace data.

- **OneShotTimer:** This can be part of an always on timer at connected standby. The OS fires OneShotTimer every 30 seconds. Applications can create a timer by calling `SetTimer` or `SetEvent`.
- **PeriodicTimer:** These timers fire after the specified amount of time has elapsed, and then reset themselves to fire again.

Periodic timers are application specific and can cause kernel-mode transition while OneShotTimers are operating system specific during connected standby.

Developer should run a minimum of two tests – baseline (without apps installed) and target (with app installed) to isolate the impact of the application.

**How to collect the trace**

- Run `powercfg.exe -a` to confirm that your system supports connected standby.
• Start the trace collection by creating the batch file using following command line:

    xperf -on
    PROC_THREAD+LOADER+INTERRUPT+DPC+CSWITCH+IDL...-
    -clocktype perfcounter -buffering -buffersize 1024
    -MinBuffers 128 -MaxBuffers 128

    • PROC_THREAD+LOADER: Provides information on device interrupts and timer.
    • INTERRUPT: Useful for break event analysis. Provides information related to HW interrupts.
    • DPC: Useful for break sources analysis. Provides information related to DPC logs.
    • CSWITCH: Useful for break sources analysis. Provides information related to context switches.
    • IPI: Provides information related to inter-processor interrupts.
    • TimerSetPeriodic+TimerSetOneShot: Required stacks for timer analysis and device interrupt analysis.

• Let the system enter connected standby state (e.g. by pressing power button)
  • Wait while xperf collects the trace for a minimum of 4 hours. Long durations provide better understanding of software activity at connected standby.
• Wake the system from connected standby (e.g. by pressing power button).

• Stop the trace.

    xperf -flush
    xperf -stop
    xperf -merge \kernel.etl MyTrace.etl

Once the trace is complete, it will generate a Mytrace.etl file in the current directory.

Post processing the trace

Run this command to post process the trace file with wakeup information:

    xperf -symbols -i mytrace1.etl -o cleanCS_diag.csv -a energydiag -verbose

You can post process a selected region of the trace by adding range

    Xperf -symbols -l mytrace1.etl -o cleanCS_diag.csv -a energygdiag -range T1 T2
    where T1 and T2 are values in usec.

    e.g: xperf -symbols -i EnergyDiag.csv -a energydiag -verbose -range 1000000 15000000000
Figure 5 shows files generated after post processing.

**cleanCS_diag:** Contains all the events and system wakeup activity.

**MyTrace1:** Contains the raw information of the trace.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>cleanCS_diag</td>
<td>Microsoft Excel Comma Separated Values File</td>
<td>119 KB</td>
</tr>
<tr>
<td>MyTrace1</td>
<td>Windows Performance Analyzer Trace File</td>
<td>131,072 KB</td>
</tr>
<tr>
<td>Xperf_process</td>
<td>Windows Command Script</td>
<td>1 KB</td>
</tr>
</tbody>
</table>

**Figure 5: Trace Output Example**

**cleanCS_diag:**

Post processing the collected trace generates a log including the number of interrupts from devices, the tick of timers and it buckets the results for each CPU. It also includes the frequency of device and timer wakeup activities. This post processing can also be done on traces taken during idle and active power analysis. Post processing the script helps you find the software activity impact on battery life.
Figure 6: Post Processing Script Output

The total number of device interrupts as shown in Figure 6 is the sum of total interrupt counts for all device modules in the collected trace. Total timer expiration is the subset of the total interrupts that are due to timers. During connected standby, timer expiration includes system timers, events, oneshot timer and periodictimer which are caused due to throttling.

The next step is to find the system busyness during connected standby. You can scroll down the report until you find the histogram of Busy Enter/Exit Time - Group "All CPUs". Busy Percent provides a good understanding of total platform activity in connected standby. This gives the total system busyness. The higher the busyness factor is relative to the baseline, the higher the impact is on platform power. Figure 7 shows a trace of baseline Total busy percent without the test apps running. Figure 8 shows a trace collected with multiple applications running, as well as a background service. Comparison between Figure 7 and Figure 8 shows an increase in activity by a factor of 150x due to wakeups triggered by those applications and background service.
Figure 7: Baseline Output

<table>
<thead>
<tr>
<th>Header</th>
<th>&lt; 100us</th>
<th>100us-1ms</th>
<th>1ms-5ms</th>
<th>5ms-15ms</th>
<th>15ms-16ms</th>
<th>16ms-20ms</th>
<th>20ms-50ms</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy Bucket Time (us)</td>
<td>19384 · 1</td>
<td>3707 · 17</td>
<td>1907 · 90</td>
<td>700 · 16</td>
<td>3099</td>
<td>98294</td>
<td>46577</td>
<td>12</td>
</tr>
<tr>
<td>% of Total Busy Time</td>
<td>15.72%</td>
<td>56.50%</td>
<td>2.91%</td>
<td>1.07%</td>
<td>0.65%</td>
<td>1.50%</td>
<td>7.10%</td>
<td>...</td>
</tr>
<tr>
<td>% of Total Busy Count</td>
<td>65.33%</td>
<td>34.14%</td>
<td>0.27%</td>
<td>0.02%</td>
<td>0.00%</td>
<td>0.01%</td>
<td>0.04%</td>
<td>...</td>
</tr>
<tr>
<td>Total Busy Time (ms)</td>
<td>5578</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Busy Interval (us)</td>
<td>188</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: Trace Output with Apps Installed

Analyzing the raw traces:

You can also inspect the trace file directly with Windows Performance Analyzer. Figure 9 shows the Graph Explorer in WPA.

Figure 9: WPA Window after opening a trace file
Figure 10 shows computation data in the analysis tab. You can zoom into the narrow bands of activity to see the wakeup activities from Processes and System. Figure 10 shows how OneShotTimer from the System aligns with the Process activities.

![Image of figure 10 showing high-level view of System during connected standby]

**Figure 10: High-level view System during connected standby**

To verify OneShotTimer calls from the system, drag and drop generic events from the system activity group into the Analysis tab window. Load the symbols from Microsoft server or from an application symbols directory by using “Load Symbols” from the Trace menu. Figure 11 shows the item enabled in the Trace menu.

![Image of figure 11 showing symbols loading menu]

**Figure 11: Symbols Loading**

You can enable graph and table for stack walk and process/thread decoding in WPA by clicking the first block on the right corner of WPA Graph, as shown in Figure 12.

![Image of figure 12 showing WPA Graph and Table]

**Figure 12: WPA Graph and Table**
The next step is to enable columns as shown in Figure 12 to get the stack walk on OneShotTimer.

Arrange the columns of the analysis table to find the wakeup activities from system or application services. Figure 13 shows process – “System” with threadID – “68” triggers OneShotTimer 36 times over the visible duration. Wakeup is happening every 30 seconds from the system process.

![Figure 13 WPA showing the stack walk of OneShotTimer](image)

**Good behavior vs. Bad behavior:**

Differentiating between good and bad behavior is important when optimizing applications for better connected standby battery life. Activities like storage access or network access by software updates are examples of things that can cause wakeup if it happens outside system wakeup.

**Good behavior**: An application service happens within “System” process. E.g. Application service goes to sleep before “System” process enters sleep state. This helps to meet the Microsoft Windows 8 connected standby WHQL requirement of 5% battery loss during 16 hours of system in connected standby.

**Bad behavior**: An application activity happens independent of “System” process or enters sleep state after “System” enters sleep state. Wake-ups which are not aligned can cause battery degradation in connected standby and can fail the Microsoft WHQL requirement.
Figure 14 shows good vs. bad behavior in connected standby.

**Figure 14: Good/Bad behavior in connected standby**

**Case Study 1: Storage Access**

A need to access local storage is very common for software services, such as anti-virus or software update services. When these services are running in connected standby, the local storage access should be delayed until the System process wakes up. Figure 15 shows a scenario of storage access for ~65 seconds in connected standby. The application wakes up
when “System” process (marked in orange) enters active sleep state. “ProcessX.exe” starts the storage access in “System32” which prevents the system from entering connected standby. The application can be optimized by removing the long storage access. If the application needs access storage in connected standby, it can be done by coalescing with system activity and going into the suspend state by broadcasting a power state transition notification.

![CPU Usage](image1.png)

![Disk Usage](image2.png)

Figure 15: Storage access by an App service in connected standby

Once that change is made, Figure 16 shows storage and “System” process coalesced in connected standby. This shows good behavior where the application is not impacting system power in connected standby.
Case Study 2: Application threads wakeup

Optimizing an application wakeup caused by the OS is tricky to analyze. You need to understand CPU Precise and Generic events to find if OneShotTimer is happening within the “System” process wakeup. Figure 16 shows a wakeup by the application thread when the “System” process is in a sleep state. This is bad way of writing the process services which keeps the system awake unnecessarily. ProcessX.exe (ID: 2440) creates several threads. The table in Figure 16 shows two threads are not aligned to the “System” ready process. Using the generic events table, you can map the threadID to setTimer and clock interrupts. As shown in Figure 16, there are Timer Set Thread tasks which need to be investigated (Thread ID 3432 and Thread ID 1824). The next step is to map the thread ID identified in the previous step (Thread ID 3432 and Thread ID 1824) to CPU Usage (Precise) table to find the activity associated with the threads. It can be related to either Timer Set or to thread schedule or to I/O activity. You can plot different charts in one view to visualize the issue.
Figure 17: App threads are keeping system active during a sleep state

The SetTimer function can be used to modify the thread timer in an application.

```c
UINT_PTR WINAPI SetTimer(
    _In_opt_ HWND hWnd,
    _In_ UINT_PTR nIDEvent,
    _In_ UINT uElapse,
    _In_opt_ TIMERPROC lpTimerFunc
);
```

The application window (HWND) is used to handle notification through the window procedure which been called after “uElapse” microseconds causing wakeup even after the “System” process has entered into connected standby state.

To fix this, if your application has a window (HWND) and you want to handle these notifications through the window procedure, call `RegisterSuspendResumeNotification` to register for these messages (or `UnregisterSuspendResumeNotification` to unregister). You can use DEVICE_NOTIFY_WINDOW_HANDLE in the Flags parameter, and pass your window’s HWND in as the Recipient parameter. The message received is the WM_POWERBROADCAST message.

If your application does not have a HWND handler or if you want a direct callback, call `PowerRegisterSuspendResumeNotification` to register for these messages (or `PowerUnregisterSuspendResumeNotification` to unregister). You can use
DEVICE_NOTIFY_WINDOW_HANDLE in the Flags parameter and pass a value of type PDEVICE_NOTIFY_SUBSCRIBE_PARAMETERS in the Recipient parameter.

**Conclusion**

Enabling applications for connected standby is important for battery life. Systems that support connected standby must meet Connected Standby Windows Hardware Certification (WHCK) requirements for battery life. This requirement specifies that all connected standby systems MUST drain less than 5% of system battery capacity over a 16 hour idle period in the default shipping configuration. A certification test can be found in the Microsoft WHCK.

**About the Author**

Manuj Sabharwal is a software engineer in the Software Solutions Group at Intel. Manuj has been involved in exploring power enhancement opportunities for idle and active software workloads. He has significant research experience in power efficiency and has delivered tutorials and technical sessions in the industry. He also works on enabling client platforms through software optimization techniques.

**References**


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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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.

Intel product plans in this presentation do not constitute Intel plan of record product roadmaps. Please contact your Intel representative to obtain Intel's current plan of record product roadmaps.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to:

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, Core, VTune, Ultrabook, Sponsors of Tomorrow and the Intel logo are trademarks of Intel Corporation in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright ©2013 Intel Corporation.