Windows On/Off Transition Performance Analysis

April 11, 2011

Abstract

This paper discusses the importance of on/off transition performance, methodologies for measuring this performance, and how to analyze the results. The information in this paper is intended to help original equipment manufacturers (OEMs), independent software vendors (ISVs), independent hardware vendors (IHVs), and systems analysts improve system response times.

This information applies to the following operating systems:  
 Windows® 7   
 Windows Vista®

References and resources discussed here are listed at the end of this paper.

The current version of this paper is maintained on the Web at:   
 <http://msdn.microsoft.com/en-us/windows/hardware/gg463386>

**Disclaimer**: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product.

© 2011 Microsoft Corporation. All rights reserved.

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Change |  |  |  |
| April 11, 2011 | Updated Figure 11; minor fixes to Figures 4-13; updated the disclaimer; fixed a typo; updated WHDC URLs. | | | |
| September 15, 2009 | Updated for Windows 7 with new appendix that includes XPerf walkthroughs. | | | |
| September 8, 2008 | First publication as “On/Off Transition Performance Analysis of Windows Vista” | | | |

Contents

[Introduction 5](#_Toc240696422)

[Windows On/Off Transitions 6](#_Toc240696423)

[Powering-on Transitions 6](#_Toc240696424)

[Powering-off Transitions 7](#_Toc240696425)

[Measuring On/Off Transition Performance 7](#_Toc240696426)

[Creating a Baseline Measurement 7](#_Toc240696427)

[Capturing a Trace by Using the Windows Performance Analyzer Tools 7](#_Toc240696428)

[Timing Traces and Analysis Traces 8](#_Toc240696429)

[Reducing the Variance in Test Results 9](#_Toc240696430)

[Prefetcher Effects on Boot Performance Measurements 9](#_Toc240696431)

[Network Connection Effects on Performance Measurements 10](#_Toc240696432)

[User Interaction Effects on Performance Measurements 10](#_Toc240696433)

[Other Test Notes 10](#_Toc240696434)

[Performance Analysis Overview 11](#_Toc240696435)

[Boot Transition 12](#_Toc240696436)

[Boot Transition: BIOSInitialization Phase 12](#_Toc240696437)

[Boot Transition: The OSLoader Phase 12](#_Toc240696438)

[Boot Transition: The MainPathBoot Phase 12](#_Toc240696439)

[MainPathBoot Phase: PreSMSS Subphase 13](#_Toc240696440)

[MainPathBoot Phase: SMSSInit Subphase 14](#_Toc240696441)

[MainPathBoot Phase: WinLogonInit Subphase 14](#_Toc240696442)

[MainPathBoot Phase: ExplorerInit Subphase 14](#_Toc240696443)

[Boot Transition: The PostBoot Phase 14](#_Toc240696444)

[Boot Transition Analysis: Capturing Traces 15](#_Toc240696445)

[Boot Transition Analysis: Processing Traces 15](#_Toc240696446)

[XML Summary 16](#_Toc240696447)

[Summaries of Plug and Play and Services Activity 17](#_Toc240696448)

[Boot Transition Analysis: Analyzing Traces 17](#_Toc240696449)

[Boot Transition Analysis: Phase Performance Vulnerabilities and Analysis 21](#_Toc240696450)

[BIOSInitialization Performance Vulnerabilities 21](#_Toc240696451)

[OSLoader Performance Vulnerabilities 21](#_Toc240696452)

[OSLoader Performance Analysis 22](#_Toc240696453)

[PreSMSS Performance Vulnerabilities 22](#_Toc240696454)

[PreSMSS Performance Analysis 24](#_Toc240696455)

[SMSSInit Performance Vulnerabilities 24](#_Toc240696456)

[SMSSInit Performance Analysis 25](#_Toc240696457)

[WinLogonInit Performance Vulnerabilities 26](#_Toc240696458)

[WinlogonInit Performance Analysis 27](#_Toc240696459)

[ExplorerInit Performance Analysis 28](#_Toc240696460)

[PostBoot Performance Vulnerabilities 29](#_Toc240696461)

[PostBoot Performance Analysis 29](#_Toc240696462)

[Boot Transition: Summary 30](#_Toc240696463)

[Sleep and Hibernate Transitions 30](#_Toc240696464)

[Overview of the Suspend Phase 31](#_Toc240696465)

[Overview of the Resume Phase 31](#_Toc240696466)

[Sleep and Hibernate Transitions: Capturing Traces 32](#_Toc240696467)

[Sleep and Hibernate Transitions: Analyzing Traces 32](#_Toc240696468)

[Sleep and Hibernate Transitions: SuspendApps Subphase 34](#_Toc240696469)

[What Happens in SuspendApps 34](#_Toc240696470)

[SuspendApps Performance Vulnerabilities 34](#_Toc240696471)

[SuspendApps Performance Analysis 34](#_Toc240696472)

[Sleep and Hibernate Transitions: SuspendServices Subphase 35](#_Toc240696473)

[What Happens in SuspendServices 35](#_Toc240696474)

[SuspendServices Performance Vulnerabilities 35](#_Toc240696475)

[SuspendServices Performance Analysis 35](#_Toc240696476)

[Sleep and Hibernate Transitions: QueryDevices and SuspendDevices Subphases 36](#_Toc240696477)

[What Happens in QueryDevices and SuspendDevices 36](#_Toc240696478)

[Sleep and Hibernate Transitions: ResumeDevices Subphase 37](#_Toc240696479)

[What Happens in ResumeDevices 37](#_Toc240696480)

[ResumeDevices Performance Vulnerabilities 37](#_Toc240696481)

[ResumeDevices Performance Analysis 38](#_Toc240696482)

[Sleep and Hibernate Transitions: ResumeServices and ResumeApps Subphases 38](#_Toc240696483)

[What Happens in ResumeServices and ResumeApps 38](#_Toc240696484)

[Sleep and Hibernate Transitions: PostResume Subphase 39](#_Toc240696485)

[What Happens in PostResume 39](#_Toc240696486)

[Sleep and Hibernate Transitions: Summary 39](#_Toc240696487)

[Shutdown Transition 39](#_Toc240696488)

[Shutdown Transition: Capturing Traces 40](#_Toc240696489)

[Shutdown Transition: Analyzing Traces 40](#_Toc240696490)

[Shutdown Transition: UserSession Phase 41](#_Toc240696491)

[What Happens in UserSession 41](#_Toc240696492)

[UserSession Performance Vulnerabilities 42](#_Toc240696493)

[UserSession Performance Analysis 43](#_Toc240696494)

[Shutdown Transition: SystemSession Phase 44](#_Toc240696495)

[What Happens in SystemSession 44](#_Toc240696496)

[SystemSession Performance Vulnerabilities 44](#_Toc240696497)

[SystemSession Performance Analysis 45](#_Toc240696498)

[Shutdown Transition: KernelShutdown Phase 46](#_Toc240696499)

[What Happens in KernelShutdown 46](#_Toc240696500)

[Shutdown Transition: Summary 47](#_Toc240696501)

[Conclusions 47](#_Toc240696502)

[Resources 48](#_Toc240696503)

[Appendix A: Identifying On/Off Performance Issues by Using Xperf 49](#_Toc240696504)

[Boot: Identifying Boot and System Plug and Play Issues 49](#_Toc240696505)

[Boot: Identifying Service Start Time Issues 53](#_Toc240696506)

[Sleep and Hibernate Transitions: Identifying Driver Delays in the SuspendDevices and ResumeDevices Phases 57](#_Toc240696507)

[Shutdown Phase: Measuring Application Delays in the UserSession Subphase 62](#_Toc240696508)

[All Transitions: Identifying Applications That Consume Significant CPU Time 64](#_Toc240696509)

[Appendix B. Raw Event Dumps 67](#_Toc240696510)

[Appendix C. .CSV Summaries of Plug and Play and Service Activity 69](#_Toc240696511)

# Introduction

Good performance during Windows® on/off transitions—boot, sleep, and shutdown—is critical for a good user experience because it improves the perceived quality of a computer and provides consistent on/off transitions.

Transition performance is important for success in the following customer scenarios:

* Users consider the time that is required to boot a computer to be a primary benchmark of hardware and operating system performance.
* System updates protect Windows users and introduce new features, but users are frustrated by lengthy update and reboot cycles. A lengthy shutdown time can irritate users and can increase service times for system administrators. Longer shutdown times can also adversely affect the reliability of mobile systems. For example, they increase the risk of power being switched off unexpectedly.
* Mobile users rely on their computer’s ability to quickly transition to sleep. Sleep performance is critical to maintaining data integrity. If resuming from sleep takes too long, users ignore sleep and shut down their computer instead.

This paper explains the Windows on/off transitions in detail, highlights performance vulnerabilities within each transition, and shows how to identify and analyze these issues by using the Windows Performance Toolkit (WPT). Performance analysis is often necessary because system extensions such as applications, drivers, services, and devices can negatively affect on/off transition times if they are not properly optimized. Poorly optimized system extensions usually result in the following:

* Delays
* Lack of parallelism
* Excessive resource consumption

The guidance in this paper can help significantly reduce on/off transition times. We have applied these performance optimizations to many systems in our laboratories and reduced boot-to-desktop time on some systems by almost 50 percent. On some systems, boot time decreased by a total of 40 to 50 seconds. However, the effect of each driver, service, or application on transition times is unique, and your results might differ.

Table 1 shows Windows 7 boot times that we measured on four real-world machine configurations. This data shows the elapsed time from the end of BIOS until the end of the post-boot phase, when the system becomes fairly idle. Boot phases are further described in “Boot Transition” later in this document.

Table 1. Optimized Boot Times on Machines Running Windows 7

|  |  |  |
| --- | --- | --- |
| System type | General specifications | Optimized boot time (seconds) |
| High-performance desktop | Dual-core 2.8-gigahertz (GHz) CPU  10,000-RPM disk  3-gigabyte (GB) memory | 10.0 |
| Thin laptop, solid-state disk (SSD) | Dual-core 2-GHz CPU  SSD  3-GB memory | 12.2 |
| Thin laptop | Dual-core 2.4-GHz CPU  5,400-RPM disk  3-GB memory | 19.3 |
| Netbook | 1.6-GHz CPU  5,400-RPM disk  1-GB memory | 33.5 |

Although these systems have good boot times, each one can still benefit from proprietary device and driver improvements. For example, booting the Windows operating system requires that a large amount of data be read from the machine’s hard disk. This data is not always sequential. Because frequent disk accesses can delay boot, good boot performance requires hard disk I/O optimizations.

For a list of best practices related to on/off transitions, see “Windows On/Off Transitions Solutions Guide” on the WHDC Web site.

# Windows On/Off Transitions

The Advanced Computer Power Interface (ACPI) specification defines system power states, or S-states. S‑states are colloquially called on/off states because the most common S-state transition is a computer turning on and off. A computer that is running Windows can transition between on/off states in the following ways:

* Booting
* Shutting down
* Entering or resuming from sleep
* Entering or resuming from hibernate

Each transition is vulnerable to performance issues. Some issues are general to all transitions, and some are specific to each transition.

## Powering-on Transitions

The powering-on transitions are as follows:

* Boot
* Resume from sleep
* Resume from hibernate

Powering-on transitions consist of two main phases:

* The first phase includes the execution of critical path code that is required to create the desktop process when the system is booting or to resume the devices on the system when the system is resuming from sleep or hibernate.
* The second phase corresponds to the time that is required for the system to reach an idle state, when background service initialization is complete. This is known as the PostBoot phase or the PostResume subphase.

Each transition, phase, and subphase has unique challenges and vulnerabilities.

## Powering-off Transitions

The powering-off transitions are as follows:

* Suspend to sleep (S3)
* Suspend to hibernate (S4)
* Shutdown (S5)

Powering-off begins when the transition is initiated and ends when the system has fully entered the S3, S4, or S5 state.

# Measuring On/Off Transition Performance

You can simplify measuring the performance of on/off transitions by using an accurate and repeatable testing methodology. A consistent approach and common tool set make it easier to identify and troubleshoot on/off performance issues.

## Creating a Baseline Measurement

Before you measure performance, you should first obtain a baseline measurement. Measure the baseline on a retail installation of Windows that has up-to-date drivers for all devices but no additional features or applications. Retail installation refers to a clean installation of the Windows operating system with no additional system extensions such as external devices, applications, or services.

After you obtain a baseline measurement, add devices, applications, and other system extensions one at a time so that you can evaluate their effects on on/off transition times compared to the baseline measurement.

## Capturing a Trace by Using the Windows Performance Analyzer Tools

The Windows Performance Analyzer tool suite, which is part of the WPT, includes tools that you can use to measure on/off transition performance. These tools rely on Event Tracing for Windows (ETW) to log events for disk I/O, CPU usage, Plug and Play, and other key components. Table 2 lists the tools in the WPT.

Table 2. Windows Performance Analyzer Command-Line Tools

|  |  |
| --- | --- |
| Tool | Purpose |
| **Xbootmgr.exe** | The Xbootmgr tool controls the state transition test environment. It automates the collection of reliable on/off performance traces. Xbootmgr also initializes ETW tracing so that data can be captured during transitions. |
| **Xperf.exe** | During on/off transition analysis, Xperf performs post processing. It generates summary reports in XML for all on/off transitions and provides raw event dumps for detailed investigations. |
| **Xperfview.exe** | Xperfview provides a detailed visualization of performance data. |

The first step to identify performance issues is to take traces on a baseline system and a modified system and then compare the results for each phase.

The trace capture and analysis process is as follows:

1. Use Xbootmgr to capture the trace during the on/off transition.

2. Use Xperf to filter the trace data and create an XML summary or a raw data file.  
 -or-  
Use Xperfview to create a graphical representation of the trace data to aid analysis.

Table 3 lists the Xbootmgr and Xperf options that you should use to capture and analyze traces for each transition.

Table 3. Xbootmgr and Xperf Command-Line Options for Trace Capture and Analysis

|  |  |  |
| --- | --- | --- |
| Power state transition to examine: | To create the trace, type:   xbootmgr  followed by: | To process the trace, type:   xperf –i *filename*  followed by: |
| Boot | **-trace** boot | **-a** boot |
| Shutdown | **-trace** shutdown | **-a** shutdown |
| Suspend and resume | **-trace** standby | **-a** suspend |
| Hibernate and resume | **-trace** hibernate | **-a** suspend |
| Boot and shutdown cycle | **-trace** rebootCycle | **-a** boot  **-a** shutdown (on two separate command lines) |

Examples and useful command-line options for each state transition appear in their respective sections later in this paper.

## Timing Traces and Analysis Traces

Timing traces provide an excellent low-impact way to measure on/off transition performance on a system. To collect timing traces, configure Xbootmgr to enable only a small set of instrumentation.

For a timing trace, specify the **-traceFlags** argument as follows:

**-traceFlags base+cswitch+compact\_cswitch**

The **base** flag represents a group of events that correspond to the basic set of kernel instrumentation that is required to measure the performance of the transition. The **cswitch+compact\_cswitch** flag specifies the collection of CPU context switching data that is required to account for CPU utilization during one of the boot phases. This information is required to determine the length of the final boot phase.

Unlike timing traces, analysis traces require a significant amount of instrumentation to be enabled on the system. The instrumentation allows more efficient analysis of the causes of performance issues, but often results in a large trace file.

Use the following Xbootmgr options to collect analysis traces:

* **-traceFlags latency+dispatcher**
* **-stackWalk Profile+CSwitch+ReadyThread**

Because timing traces use minimal instrumentation, it can be difficult to determine the causes of performance issues that appear in timing traces. Timing traces are appropriate for measuring the performance of on/off transitions, whereas analysis traces are more appropriate for analyzing on/off transition performance issues.

## Reducing the Variance in Test Results

To ensure the reliability of performance test data, you must minimize variance. The Xperf and Xbootmgr tools capture many system events, so the key to minimizing variance is controlling system activity. To minimize noise during testing, you must eliminate or carefully control variable attributes of the system such as the boot prefetcher, the status of network connections, whether the logon process is manual or automatic, and any manual user interactions.

### Prefetcher Effects on Boot Performance Measurements

Booting the Windows operating system requires a large amount of data to be read from the machine’s hard disk. The Windows *boot prefetcher*—also called ReadyBoot—optimizes the boot process by using the disk I/O pattern from previous boots to predict the I/O pattern of the next boot. In effect, each boot “trains” Windows to accurately predict what data will be needed. The boot prefetcher can then read data from the hard disk before the data is actually required, which improves boot performance.

We also refer to the sequence of prefetched data as the *boot plan*. The prefetcher heuristics are designed so that the boot plan grows more accurate as additional training boots are performed. An accurate boot plan means that ReadyBoot can prefetch the correct data into memory before Windows needs it. This reduces the time that Windows waits on disk I/O during the boot process. Typically, end users do not boot untrained systems because the design ensures that the system is trained after only a few boots.

Table 4 shows the effects of the boot prefetcher in Windows Vista® on a dual-core, 2.66‑GHz system with 3 GB of RAM and a 7,200‑RPM hard disk. The “Untrained” entry in the table shows the boot time for an untrained system, followed by the timing results for five consecutive boots.

Table 4. Boot Prefetcher Effects on Windows Vista Boot Times

|  |  |
| --- | --- |
|  | Time to boot completion  (in milliseconds—ms) |
| Untrained | 47,538 |
| Boot 1 | 24,906 |
| Boot 2 | 18,765 |
| Boot 3 | 16,953 |
| Boot 4 | 17,277 |
| Boot 5 | 17,239 |

To reduce variance in test measurements, use the **‑prepsystem** command-line option to Xbootmgr.exe so that the prefetcher can fully self-optimize. This feature of the WPT automates training boots during transition testing to ensure that all Windows Vista and Windows 7 optimizations are performed.

### Network Connection Effects on Performance Measurements

The presence of a network connection can influence the performance of on/off transitions. To reduce the variability that networking can introduce, follow these recommendations:

* Leave wireless adapters enabled but not connected to any network. Network connections can affect resume performance. If connections must be established during the transition, be sure to account for connection establishment delays during analysis.
* Delete all persistent network connections before you start testing, by using the following command from an elevated command line:

net \* /d

* When the test system is joined to a domain, the group policies that the domain administrator deploys can affect performance. Therefore, consider group policies when you design your test scenarios.

### User Interaction Effects on Performance Measurements

In addition to testing-related activities, system activities that are unrelated to testing appear in ETW traces. For best results, do not interact with the test systems during test runs, because the system activity could distort the test results.

To minimize system activity and enable test automation, add the user account that runs Xbootmgr.exe to the Administrators group, disable user account control (UAC), and configure user profiles for auto-logon. This avoids otherwise necessary activity and user interaction.

You can perform most of these tasks by using User Accounts in Control Panel. To change some of these settings, you must have administrative credentials.

## Other Test Notes

The following additional notes apply to running Xbootmgr.exe:

* Test-signed drivers can affect boot times. If any test-signed drivers are installed on the system, total boot time can increase by up to 8 seconds, depending on system configuration.
* Code integrity checks occur after new driver installation. If you install new drivers, let the systems idle for 15 minutes before you start a training run to ensure that code integrity checks do not interfere with results.
* Countdown dialog boxes appear during pauses between intervals of the test process.
* Transitions to S3 and S4 are supported only when the **On resume, display logon screen** check box under **Screen Saver Settings** is cleared.
* By default, Xbootmgr.exe saves results in the directory from which it was started. To changes this default, use the **–resultPath** argument to specify a different location.
* Copy the \*.etl files to a separate Windows system for processing. The trace files can be large, and processing them between test runs can alter the state of the test machine.

# Performance Analysis Overview

The following sections discuss the analysis of the following on/off transitions:

* Boot
* Sleep and Hibernate
* Shutdown

For each transition type, we will:

* Look at the commands that are required to generate a performance trace.
* Examine the transition thoroughly and indicate potential vulnerabilities.
* List the XML nodes to which you should pay particular attention in the trace summary.

Each on/off transition consists of multiple phases and subphases, as Figure 1 shows.



Figure 1. Overview of the phases and subphases associated with on/off transitions

# Boot Transition

Operating system initialization and device and driver initialization involve a lot of code and complicated interaction. Because system resources are taxed during boot, reducing resource usage as much as possible is critical to eliminate bottlenecks and improve performance.

The boot transition can be divided into four high-level phases that are shown in Figure 2. A description of each phase is given, followed by a walkthrough of boot analysis.



Figure 2. High-level phases of the boot transition

## Boot Transition: BIOSInitialization Phase

#### What Happens in This Phase

During the BIOSInitialization phase, the platform firmware identifies and initializes hardware devices, and then runs a power-on self-test (POST). The POST process ends when the BIOS detects a valid system disk, reads the master boot record (MBR), and starts Bootmgr.exe. Bootmgr.exe finds and starts Winload.exe on the Windows boot partition, which begins the OSLoader phase.

#### Visual Cues

The BIOS splash screens and any POST-related messages appear during BIOSinitialization.

## Boot Transition: The OSLoader Phase

#### What Happens in This Phase

During the OSLoader phase, the Windows loader binary (Winload.exe) loads essential system drivers that are required to read minimal data from the disk and initializes the system to the point where the Windows kernel can begin execution. When the kernel starts to run, the loader loads into memory the system registry hive and additional drivers that are marked as BOOT\_START.

#### Visual Cues

This phase begins approximately when the BIOS splash and diagnostic screens are cleared and ends approximately when the “Loading Windows” splash screen appears.

## Boot Transition: The MainPathBoot Phase

#### What Happens in This Phase

During the MainPathBoot phase, most of the operating system work occurs. This phase involves kernel initialization, Plug and Play activity, service start, logon, and Explorer (desktop) initialization. To simplify analysis, we divide the MainPathBoot phase into four subphases, as Figure 3 shows. Each subphase has unique characteristics and performance vulnerabilities.

#### Visual Cues

Visually, the MainPathBoot phase begins when the “Starting Windows” splash screen appears and lasts until the desktop appears. If auto-logon is not enabled, the time that elapses while the logon screen is displayed affects the measured boot time in a trace.

Figure 3. The MainPathBoot subphases

### MainPathBoot Phase: PreSMSS Subphase

#### What Happens in This Subphase

The PreSMSS subphase begins when the kernel is invoked. During this subphase, the kernel initializes data structures and components. It also starts the PnP manager, which initializes the BOOT\_START drivers that were loaded during the OSLoader phase.

When the PnP manager detects a device, it loads and initializes the device’s drivers in the following sequence:

1. Detects a device.

2. Loads the drivers into memory and validates the driver signature.

3. Calls the **DriverEntry** function of the device driver. The driver code executes.

4. Sends an IRP\_MN\_START\_DEVICE I/O request packet (IRP) to the driver, which notifies the driver to start the device. The driver code executes.

5. Sends the IRP\_MN\_QUERY\_DEVICE\_RELATIONS IRP to the driver to enumerate any child devices. The driver code executes.

The PnP manager repeats these steps for each enumerated child device. It continues to iterate through child devices until it has enumerated and initialized the full device tree.

#### Visual Cues

PreSMSS begins approximately when the “Loading Windows” splash screen appears. There are no explicit visual cues for the end of PreSMSS.

### MainPathBoot Phase: SMSSInit Subphase

#### What Happens in This Subphase

The SMSSInit subphase begins when the kernel passes control to the session manager process (Smss.exe). During this subphase, the system initializes the registry, loads and starts the devices and drivers that are not marked BOOT\_START, and starts the subsystem processes. SMSSInit ends when control is passed to Winlogon.exe.

#### Visual Cues

There are no explicit visual cues for the start of SMSSInit, but the blank screen that appears between the splash screen and the logon screen is part of SMSSInit. It ends before the logon screen appears.

### MainPathBoot Phase: WinLogonInit Subphase

#### What Happens in This Subphase

The WinLogonInit subphase begins when SMSSInit completes and starts Winlogon.exe. During WinLogonInit, the user logon screen appears, the service control manager starts services, and Group Policy scripts run. WinLogonInit ends when the Explorer process starts.

#### Visual Cues

WinLogonInit begins shortly before the logon screen appears. It ends just before the desktop appears for the first time.

### MainPathBoot Phase: ExplorerInit Subphase

#### What Happens in This Subphase

The ExplorerInit subphase begins when Explorer.exe starts. During ExplorerInit, the system creates the desktop window manager (DWM) process, which initializes the desktop and displays it for the first time.

This phase is CPU intensive. The initialization of DWM and desktop occurs in the foreground, while in the background the service control manager (SCM) starts services and the memory manager prefetches code and data. On most systems ExplorerInit is CPU bound, and timing issues are likely the result of a simple resource bottleneck.

#### Visual Cues

ExplorerInit begins just before the desktop appears for the first time. There is no clear visual cue to indicate the end of ExplorerInit.

## Boot Transition: The PostBoot Phase

#### What Happens in This Phase

The PostBoot phase includes all background activity that occurs after the desktop is ready. The user can interact with the desktop, but the system might still be starting services, tray icons, and application code in the background. This phase is considered complete when Xperf data indicates that background activity has dropped to a reasonably idle level.

Specifically, Xperf samples the system every 100 ms during the PostBoot phase. If the system is 80-percent or more idle (excluding low-priority CPU and disk activity) at the time of the sample, Xperf considers the system to be “idle” for that 100‑ms interval. The phase persists until the system accumulates 10 seconds of idle time.

**Note:** When you review traces and report timing results, you should subtract the 10‑second idle time that accumulated during PostBoot to determine total boot time. Busy time in PostBoot counts toward the total, but the mandatory 10 seconds of idle time does not. In this paper, the idle time is subtracted from the timing data.

The PostBoot phase can end while the disk is busy with low-priority activity. Therefore, the disk activity LED is not a good indicator of phase completion.

#### Visual Cues

There are no explicit visual cues for PostBoot. The phase begins after the user’s desktop appears and ends after satisfying the 10-second metric that was explained earlier.

## Boot Transition Analysis: Capturing Traces

The first step in boot transition analysis is to capture a trace by using the Xbootmgr tool. As mentioned earlier, Xbootmgr options control the test environment and the level of detail in the trace.

To obtain a list of Xbootmgr commands, type the following command in an elevated command prompt:

Xbootmgr –help

For complete information about Xbootmgr options, see “On/Off Transition Trace Capture Tool Reference” on the MSDN® Web site.

“Timing Traces and Analysis Traces” earlier in this paper explains the difference between timing and analysis traces. Consider an analysis boot trace as an example. From an administrative command prompt, enter the following command:

xbootmgr -trace boot -numRuns 3 -resultPath %systemdrive%\traces -postBootDelay 180 -traceFlags latency+dispatcher -stackWalk Profile+CSwitch+ReadyThread -prepsystem

In response, the following events occur:

* The test system reboots within 5 seconds after you press ENTER.
* The **-prepSystem** option causes the Windows prefetcher to optimize itself by booting the system several times. A status dialog box appears during each boot to inform you of the current progress.
* The system restarts again, and Xbootmgr takes three traces during the boot process. It writes the traces to the folder that was specified in the **–resultPath** option—in this case, the Traces folder on the system drive.

## Boot Transition Analysis: Processing Traces

The Xperf.exe and Xperfview.exe tools in WPT can help you analyze the boot transition performance data. Xperf creates summary files for simplified analysis, and Xperfview displays trace data in graphical form.

For details about Xperf and Xperfview, see the “Windows Performance Analyzer Command Line Reference” on MSDN.

### XML Summary

To generate an XML summary of the boot trace, use the **-a boot** action with Xperf. For example, the following command takes as input the Trace.etl file and generates the Summary.xml output file:

xperf -i trace.etl -o summary.xml -a boot

It is easiest to view the XML file in a reader that lets you collapse and expand XML nodes dynamically. In this paper, we use Internet Explorer to examine the Summary.xml file.

Open the Summary.xml file in Microsoft® Internet Explorer. If the gold Information Bar appears, click it and then click **Allow blocked content** to enable dynamic node expansion.

You should see XML output that resembles Figure 4.

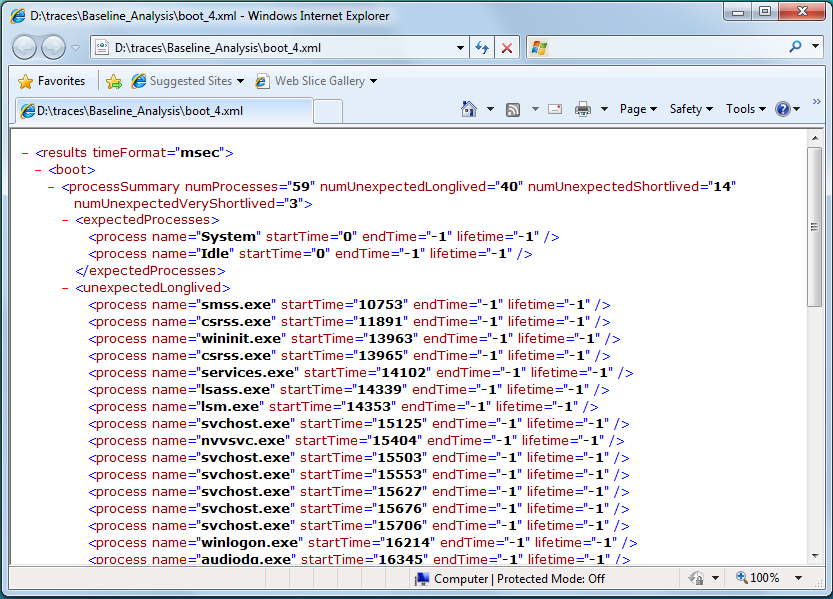


Figure 4. Summary.xml output viewed in Internet Explorer

When you examine the XML report, remember the following:

* The time unit appears in the **timeFormat** line at the top of the XML summary report. In Figure 4, the time unit is **msec**, for milliseconds.
* Most intervals and operations are described by a trio of start time, end time, and lifetime. If a process starts during the trace and is still running at the end of the trace, both the end time and the lifetime appear as -1.

### Summaries of Plug and Play and Services Activity

Many boot performance problems are caused by Plug and Play devices and drivers or services. One way to analyze this data is to create a scenario-specific, comma-separated value (.csv) summary, which you can read and manipulate by using Microsoft Excel® and similar tools. With a few exceptions, this paper does not use .csv files. For more information on how to generate these summaries, see Appendix C.

## Boot Transition Analysis: Analyzing Traces

The first step in any boot transition performance analysis is to compare overall timing metrics of a modified system to a baseline system.

Beginning with the Summary.xml file that appeared earlier in Figure 4, we collapsed most of the nodes so that the high-level summary nodes are visible. Figure 5 shows the high-level entries in the **boot** node.

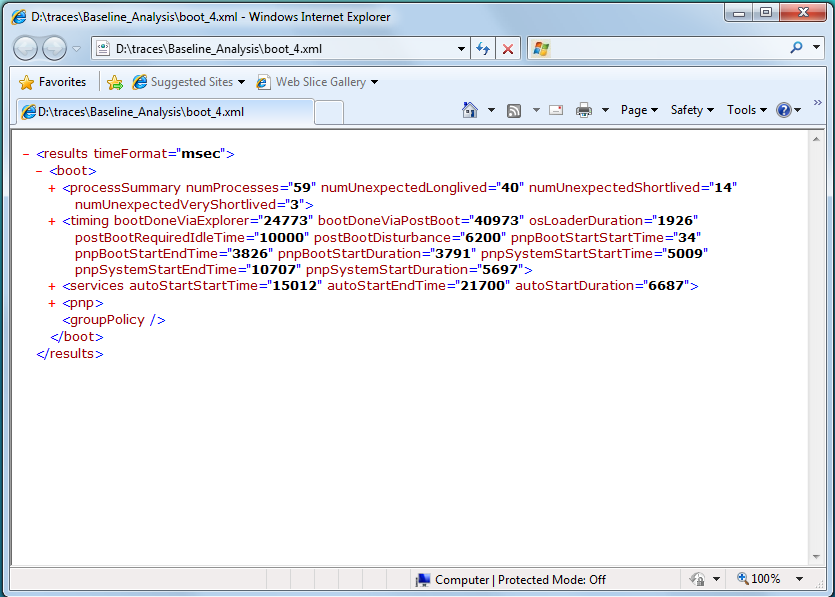


Figure 5. Summary.xml output with nodes collapsed

The following are the entries in the **boot** node:

* **ProcessSummary** contains overview performance data about the processes that run during the boot transition and acts as a parent node for process timing data.
* **Timing** summarizes transition time performance data and acts as a parent node for most of the boot transition phase summary data.
* **Services** contains overview performance data for services that the SCM starts during boot and is a parent node for service-specific data.
* **PNP** is a parent node for Plug and Play driver and device performance data such as enumeration and start times.
* **GroupPolicy** is a parent node for Group Policy script performance data, if any.

To begin to analyze boot performance, you should determine the overall boot time and the relative **phase times** for the phases that were described earlier in this paper for both the baseline and modified trace.

If you expand the **timing** node, you should see output that resembles Figure 6.

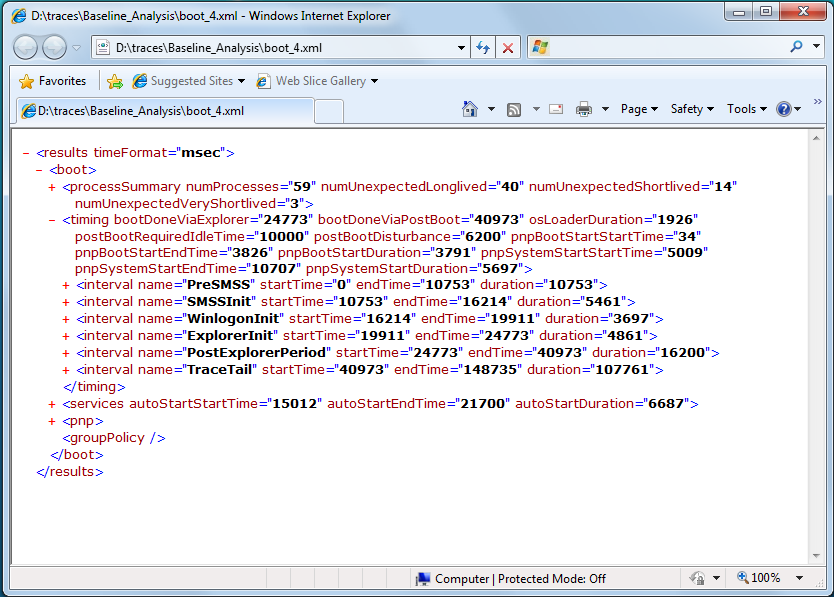


Figure 6. Summary.xml output with expanded timing node

In the preceding data, the **bootDoneViaExplorer** key shows the duration of the boot (in milliseconds) until the start of Explorer.exe. In this case, it is 24.773 seconds. As mentioned earlier in the “Boot Transition: Post Boot Phase” section, the PostBoot time is measured by using a special metric. The **bootDoneViaExplorer** figure excludes the PostBoot time.

The **bootDoneViaPostBoot** key, which appears immediately following the **bootDoneViaExplorer** key, is the length of the boot transition including PostBoot. This metric represents the total time of a boot transition. In Figure 6, the total boot time is 40.973 seconds, or30.973 seconds if the 10 seconds of PostBoot wait time is subtracted. Over 30 seconds for a boot is not especially good. This system can be optimized to boot more quickly. If you have taken a baseline trace in addition to a trace on a modified system, compare the **bootDoneViaPostBoot** values minus the 10‑second wait to determine the effects of system modifications on performance.

If the results for the baseline and modified systems differ significantly—that is, by several seconds—the next step is to examine the individual phase times to determine whether the increase occurred in one specific phase. By comparing interval durations between the modified and baseline systems, you can more deeply examine performance issues.

The main **timing** node contains OSLoader phase timing data. Timing data for other phases appears in the **interval** subnodes under the **timing** node. Each **interval** node contains the name of the interval, a start time, an end time, and a duration. The PostExplorerPeriod node contains the PostBoot time.

The **TraceTail** interval contains the rest of the trace data that was recorded after the 10-second PostBoot criterion was fulfilled, so you can generally ignore it.

On correctly trained systems, boot performance is generally consistent. However, occasionally times vary from run to run on identical machines because of nondeterministic components of the boot process. To illustrate a comparison of boot and phase times with variance, we generated another trace on the machine with no changes. The resulting Summary.xml file is shown in Figure 7.

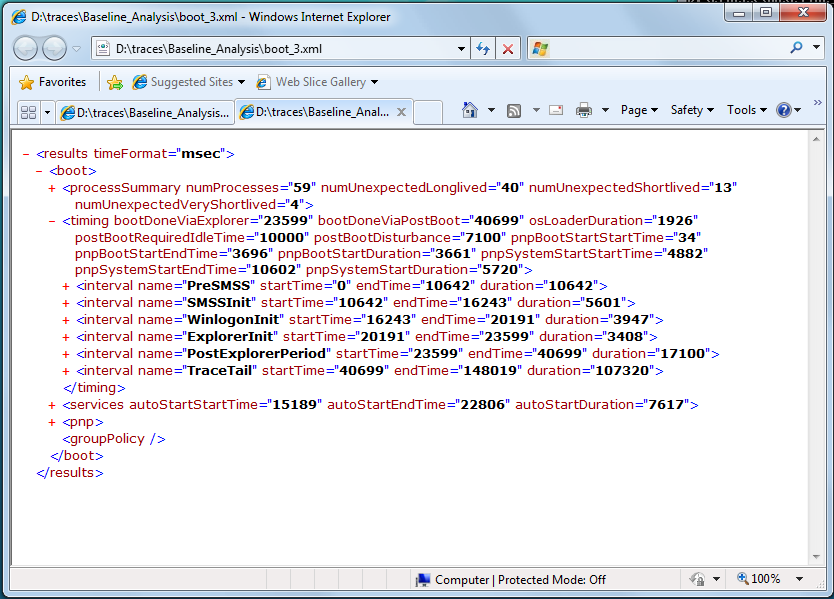


Figure 7. Summary.xml output for second boot

Table 5 compares the boot and phase times for the two trace runs, which are named Run 1 and Run 2. The boot plan was optimized before the traces were run.

Table 5. Boot and Phase Times for Two Boot Traces on the Same Hardware

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Phase or subphase | Run 1 | Run 2 | Difference | %Delta |
| OSLoader | 1926 | 1926 | 0 | 0 |
| PreSMSS | 10753 | 10642 | 111 | 1.0% |
| SMSSInit | 5461 | 5601 | -140 | -2.6% |
| WinLogonInit | 3697 | 3947 | -250 | -6.8% |
| ExplorerInit | 4861 | 3408 | 1453 | 29.9% |
| PostBoot | 16200 | 17100 | -900 | -5.6% |
| BootDoneViaExplorer | 24773 | 23599 | 1174 | 4.7% |
| Total Boot | 30973 | 30699 | 274 | 0.7% |

As Table 5 shows, large changes that occur in individual phases can have only a small effect on boot. The data in Table 5 shows that data was likely prefetched during ExplorerInit in Run 1, but was prefetched during the PostBoot phase in Run 2. This inflated the BootDoneViaExplorer time and ExplorerInit phase time fairly significantly, but overall boot time varied by less than 1 percent.

The best way to determine whether a system modification has a distinguishable effect on boot is to run several traces. Another approach is to examine other nodes of the XML output such as individual phase nodes, the **pnp** node, or the **services** node.

Boot is a resource-intensive process, and contention for resources is common. The introduction of new drivers or services can increase resource contention and therefore prolong boot times. If you notice that one particular phase has an increased time, you can expand its timing node to view CPU and disk statistics. Figure 8 shows the expansion of the PreSMSS interval.

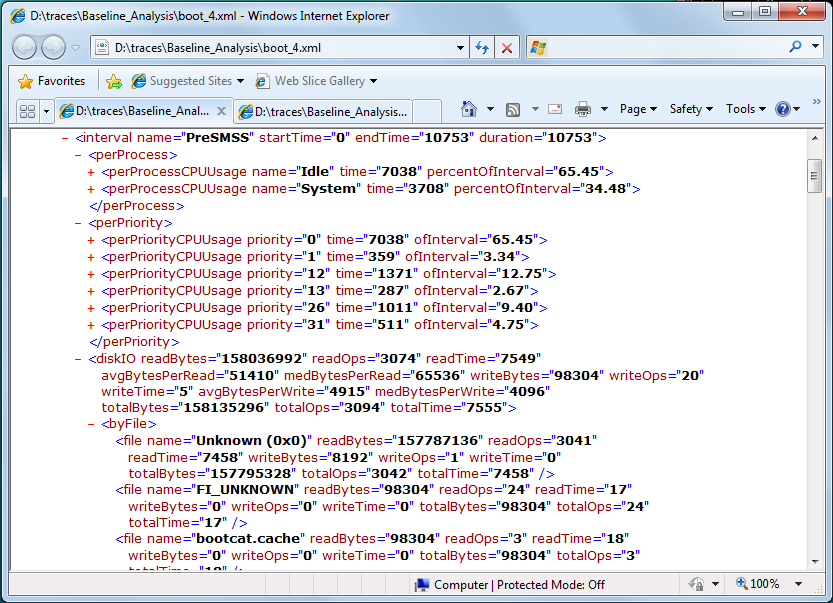


Figure 8. Summary.xml with expanded PreSMSS interval node

The following are the subnodes in the PreSMSS interval:

* **perProcess** lists all processes in order from highest CPU usage to lowest CPU usage.
* **perPriority** lists the CPU priorities and the time that the CPU spent at each priority.
* **diskIO** contains disk metrics such as the number of bytes that were read and written, the number of read and write operations, and how long was spent performing disk reads and writes. It also includes totals for each of these categories along with additional statistics. It further separates disk activity to **byFile** and **byExtension**.

These disk and CPU statistics should provide a good indication of where time and resources are being spent. The following are several points that you should note:

* The file that is named “Unknown (0x0)” dominates disk read I/O during most boot phases. This represents the boot prefetcher while it reads data into memory. You can usually ignore this file.
* Data that was prefetched for a particular process does not appear in the statistics for that process name in the PreSMSS interval. To obtain complete disk metrics for a particular process, use the FILE\_IO trace flag when you capture a boot trace and then examine the File I/O graph in Xperfview to see the complete disk metrics (reads/writes/flushes) for that process.
* The disk in Figure 8 spent more than 7.5 seconds servicing prefetcher disk reads. The CPU spent almost the same time in the idle thread, waiting on the diskIO read operations to complete.
* Devices and services that cause system waits and delays are difficult to detect through these metrics. One indication that a system extension introduces delays is an increase in CPU idle time without a corresponding increase in disk I/O time.

## Boot Transition Analysis: Phase Performance Vulnerabilities and Analysis

The next step is to dig deeper into problematic scenarios or phases that present performance issues. This section describes each phase and identifies any performance vulnerabilities.

For a list of best practices related to on/off transitions, see “Windows On/Off Transitions Solutions Guide” on the WHDC Web site.

### BIOSInitialization Performance Vulnerabilities

You must manually determine whether BIOSInitialization consumes too much time. One performance problem that is typically associated with BIOSInitialization is the time that is required for extended memory tests. Fast POST settings skip these tests and therefore reduce the time that is spent in BIOSInitialization. To minimize BIOS effects on boot transition, configure the system BIOS for fast POST, if this is possible.

### OSLoader Performance Vulnerabilities

The OSLoader phase is primarily I/O-bound. Faster disks generally lead to a faster OSLoader phase.

During the OSLoader phase of the boot transition, Windows verifies the signatures on BOOT\_START drivers. Drivers can be signed in one of two ways: catalog signing (which includes the signature in a catalog file) or embedded signing (which embeds the signature in the driver file).

BOOT\_START drivers that are catalog-signed can cause 1-second to 3‑second delays, depending on the hard disk speed. To verify a catalog-signed file, Windows must locate the catalog files on disk and load them, which reduces performance. This is a significant delay when you consider that the OSLoader phase is generally less than 3 seconds long.

The catalog load penalty does not apply for each catalog-signed driver. Windows loads the catalog files only one time. However, this shows a common theme in boot performance analysis: one badly behaving component on the critical boot path can introduce significant system-wide delays.

### OSLoader Performance Analysis

To obtain a list of the BOOT\_START drivers, type the following commands:

xperf –i trace.etl –o trace.txt –a dumper

findstr /C:"I-DCStart" trace.txt

Ensure that all drivers in this list are embedded-signed. Use SignTool.exe from the Windows Driver Kit (WDK) to verify the embedded signatures, substituting the appropriate driver name. For example:

signtool verify /v driver.sys

The OSLoader phase occurs early in the boot process. In this early stage of boot, WPT has no adequate infrastructure to track CPU or disk usage. Therefore, OSLoader does not have its own **interval** node with CPU and disk information. Instead, its duration appears in the **timing** node, as highlighted in Figure 9 on the following page.

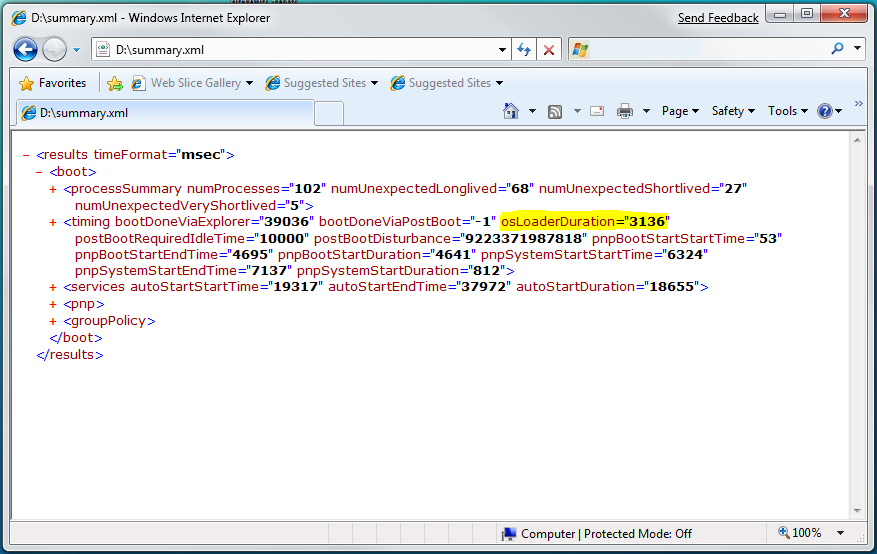


Figure 9. Location of osLoaderDuration in Summary.xml

### PreSMSS Performance Vulnerabilities

Devices and drivers can affect the time that is required for Plug and Play enumeration and initialization during the PreSMSS subphase. Some devices and drivers require lengthy initialization periods because they spend a long time processing the IRP\_MN\_QUERY\_DEVICE\_RELATIONS and IRP\_MN\_START\_DEVICEIRPs.

Long delays during IRP processing can cause significant boot performance degradations. The design of the PnP manager enables groups of devices and associated drivers to begin loading and initialization in parallel. However, all IRP\_MN\_START\_DEVICE and IRP\_MN\_QUERY\_DEVICE\_RELATIONS requests in a particular device node must be complete before the PnP manager can traverse to a subsequent node. In other words, each root device must start before devices that are attached to it can start.

Therefore, a reduction in the initialization time for a single driver or device might not reduce the overall boot time by the same amount, depending on the length of the initialization delay and whether the delay blocked node traversal. Regardless, it is always a good practice to reduce initialization time because the degree of parallelism within this subphase varies with each system configuration. Small savings on one system could represent a significant improvement on another.

Instead of processing IRP\_MN\_START\_DEVICE and IRP\_MN\_QUERY\_DEVICE\_RELATIONS directly in a dispatch routine, a driver should return STATUS\_PENDING so that the PnP manager can process other requests while the driver processes the IRP. All drivers should return STATUS\_PENDING if they can to increase parallelism and reduce boot time.

As a demonstration, we took sample runs with and without pending the IRPs in a customized driver. The driver in these tests was named Ramdisk, and the data appears in Table 6.

Table 6. Effect of Pending IRPs on Driver Initialization and Boot Times (in milliseconds)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | PreSMSS time | Plug and Play system start time | Ramdisk load time | Ramdisk start time | Ramdisk query relations time | Total boot time | Differ-ence |
| No Pended IRPs | 26636 | 20813 | 9214 | 3000 | 3000 | 44923 | Not applicable |
| Pend Start; do not pend Relations | 17919 | 13007 | 0 | 3000 | 3000 | 36839 | 8084 |
| Pend Start and Relations | 17294 | 12343 | 0 | 3000 | 3000 | 35409 | 9514 |

As the data in Table 6 shows, pending IRPs can improve boot performance significantly.

### PreSMSS Performance Analysis

The phase time for PreSMSS appears in its **interval** node, as highlighted in Figure 10. During PreSMSS, the boot Plug and Play and system Plug and Play initialization sequences occur. The **pnp** node summarizes the Plug and Play data in **phase** nodes that are named bootStart and systemStart. These are also highlighted in Figure 10.

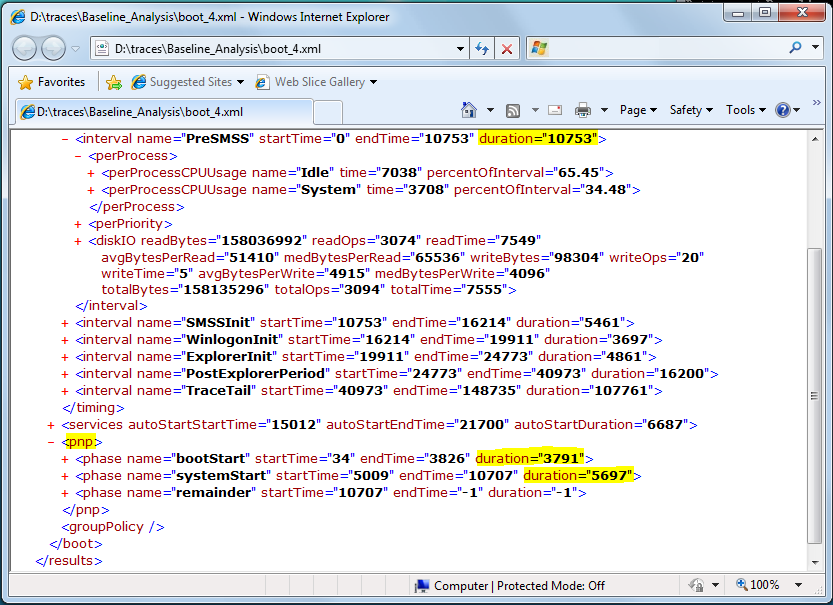


Figure 10. Location of PreSMSS duration and Plug and Play phase durations   
 in Summary.xml

Boot and system Plug and Play analysis is a detailed process. For step-by-step instructions and the details about the effects of pending on boot transition performance, see “Identifying Boot and System Plug and Play Issues” in Appendix A.

### SMSSInit Performance Vulnerabilities

Video drivers are a common source of performance problems in the SMSSInit subphase. The video driver must be initialized first in the system session and then in the user session. Reduction of video driver initialization time leads to a direct wall-clock reduction in boot time.

Initialization in the user session is typically much faster than in the system session because Windows performs common initialization tasks during the system session. You can infer the start times and end times of these initialization operations from the events that appear in the raw dump of the trace. Measuring these values is important when you evaluate an updated video driver for boot performance.

### SMSSInit Performance Analysis

The time for the SMSSInit subphase appears its **interval** node, as highlighted in Figure 11.

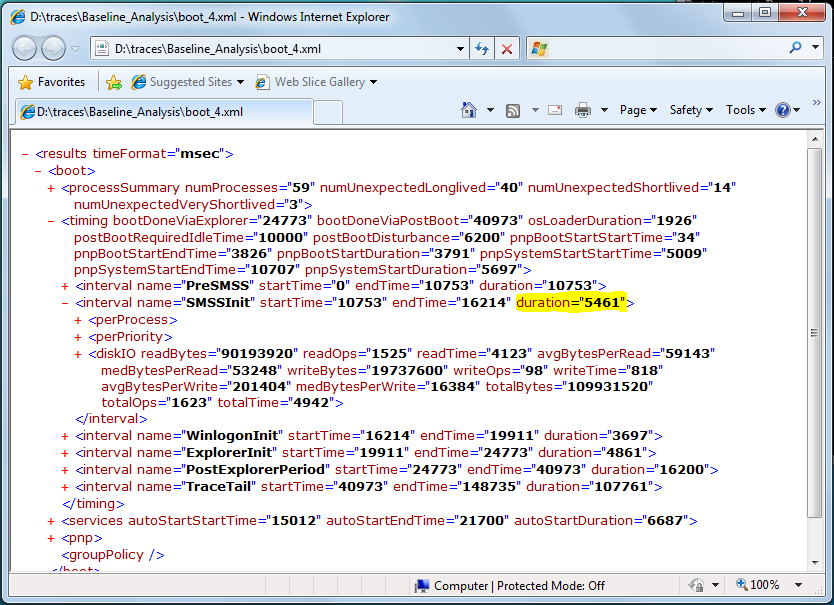


Figure 11. Location of SMSSInit duration in Summary.xml

#### Calculating the System Session Video Initialization Time

No specific events are associated with the system session video initialization process, but you can infer the time that is required by calculating the time difference between the completion of the first Csrss.exe startup by Smss.exe and the startup of Wininit.exe by Smss.exe.

The examples that follow use a .csv file. For information about creating a .csv file, see Appendix C.

To calculate the video driver initialization time in the system session

1. Filter out all SMSS events of interest:

findstr.exe /C:"Microsoft-Windows-Subsys-SMSS" trace.csv > trace.csv.filtered

findstr.exe /i “csrss wininit” trace.csv.filtered

2. Locate the end of the first start of Csrss.exe:

Event type is in the first column:  
Microsoft-Windows-Subsys-SMSS/smss:ExecuteImage/Stop

The timestamp is in the second column.

The process name is in the sixth column.

Note that two starts of Csrss.exe are expected, one per session, with the first corresponding to the system session.

3. Locate the beginning of Wininit.exe start:

Microsoft-Windows-Subsys-SMSS/smss:ExecuteImage/Start

4. To determine the time that is required for system session video initialization, calculate the difference between these two timestamps.

#### Calculating the User Session Video Initialization Time

You can infer the time that the user session video initialization process requires by noting when the second Csrss.exe startup by Smss.exe is complete and when the startup of Winlogon.exe by Smss.exe begins.

To calculate the video driver initialization time in the user session

1. Filter out all SMSS events of interest:

findstr.exe /C:"Microsoft-Windows-Subsys-SMSS" trace.csv > trace.csv.filtered

findstr.exe /i "csrss winlogon" trace.csv.filtered

2. Locate the end of the second launch of Csrss.exe.

Note that two launches of Csrss.exe are expected, one per session, with the second launch corresponding to the system session.

3. Locate the start of Winlogon.exe launch:

Microsoft-Windows-Subsys-SMSS/smss:ExecuteImage/Start

4. To determine the time that is required for user session video initialization, calculate the difference between these two timestamps.

### WinLogonInit Performance Vulnerabilities

Many operations occur in parallel during WinLogonInit. On many systems, this subphase is CPU bound and has large I/O demands. Good citizenship from the services that start in this phase is critical for optimized boot times.

Services can declare dependencies or use load order groups to ensure that they start in a specific order. Windows processes load order groups in serial order. Service initialization delays in an early load order group block subsequent load order groups and can possibly block the boot process .

We created a customized service that is called “Slowsvc” to demonstrate how service initialization delays in early load order groups can affect boot performance. This service waits for 6 seconds during its initialization routine. Table 7 shows the boot and service start times for two runs: one run includes Slowsvc in an early load order group, and the other does not.

Table 7. Effects of Service Delays in Early Load Order Groups on Service Start and Boot Times

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| System | SlowsvcInit time | WinlogonInit time | ExplorerInit time | Total services start time | Total boot time | Boot time reduc­-tion |
| Slowsvc assigned to an early load order group | 6000 | 7459 | 1822 | 10720 | 32805 | Not applicable |
| Slowsvc not assigned to a load order group | 6000 | 3893 | 2864 | 12566 | 31888 | 917 |

Long delays during any service initialization can increase the time that the boot transition requires. If you can do so, set services to demand start or trigger start. Demand-start and trigger-start services start after the boot process is complete and therefore reduce overall boot time.

The presence of persistent network connections can cause variable delays in the WinLogonInit subphase. Therefore, to ensure that each run is consistent, we recommend that you delete all persistent network connections before you test.

If the test machine is connected to a domain that deploys Group Policy, test results sometimes show that WinLogonInit requires lots of time. The actual time varies depending on the specifics of the policy or policies. See the **groupPolicy** node of the Summary.xml file to determine the effects of any Group Policy scripts on your boot times.

### WinlogonInit Performance Analysis

As with durations for the other subphases, WinlogonInit subphase duration appears in the corresponding **interval** node in the Summary.xml file.

The **services** and **groupPolicy** nodes in the Summary.xml file (which are highlighted in Figure 12) contain the performance data for individual services and Group Policy scripts, respectively, that execute during boot.

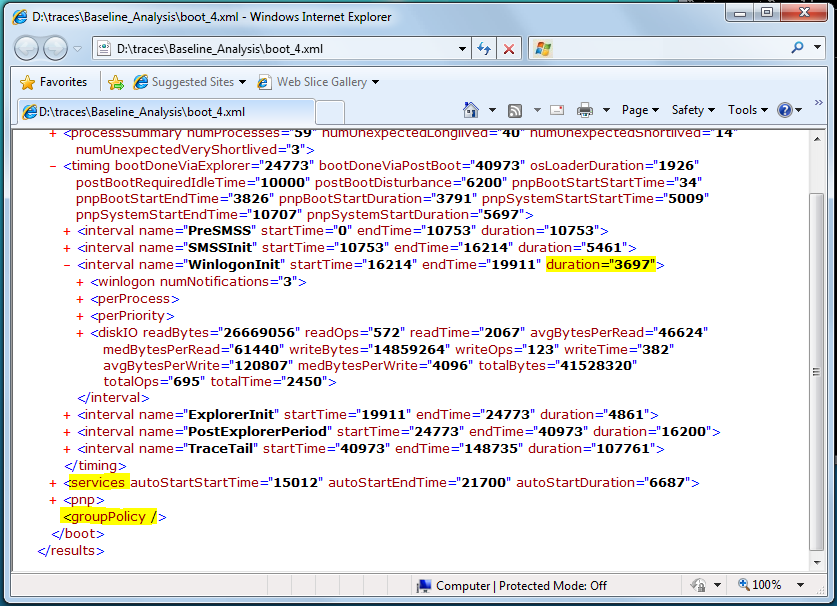


Figure 12. Location of WinLogonInit duration, Services, and GroupPolicy nodes in   
 Summary.xml

You can find information about service start timing by examining the **services** node in the Summary.xml file. Each service has its own subnode, and the time that is required for the service to begin is in the **totalTransitionTimeDelta** attribute. Or, you can use the **xperf –a services** command or the Xperfview visualization tool to identify services that have long start delays. For step-by-step instructions on how to analyze service initialization and the effects of load order groups on boot transition performance, see “Identifying Service Start Time Issues” in Appendix A.

### ExplorerInit Performance Analysis

The ExplorerInit subphase duration appears in the corresponding **interval** node in the Summary.xml file.

Applications—such as antivirus programs or application servers—that are created during service start in this or previous phases can consume CPU resources during ExplorerInit. Some services might not be started yet when ExplorerInit is complete.

Analysis of the operating system activity during this phase is complex and beyond the scope of this paper.

### PostBoot Performance Vulnerabilities

During PostBoot, Windows examines the entries in the various **Run** and **RunOnce** keys (**Run**, **RunOnce**, **RunOnceEx**, **RunServices**, and so on) in the registry and the Startup folder in the file system, and then starts the listed applications.

To improve performance, the goal for this phase should be to quantify and then limit the background processing that continues after the desktop is visible. Such processing includes running startup applications, creating tray icons, and so on.

### PostBoot Performance Analysis

The time that is spent in the PostBoot phase appears in the **interval** node that is named **PostExplorerPeriod**. The duration of this phase is highlighted in Figure 13.

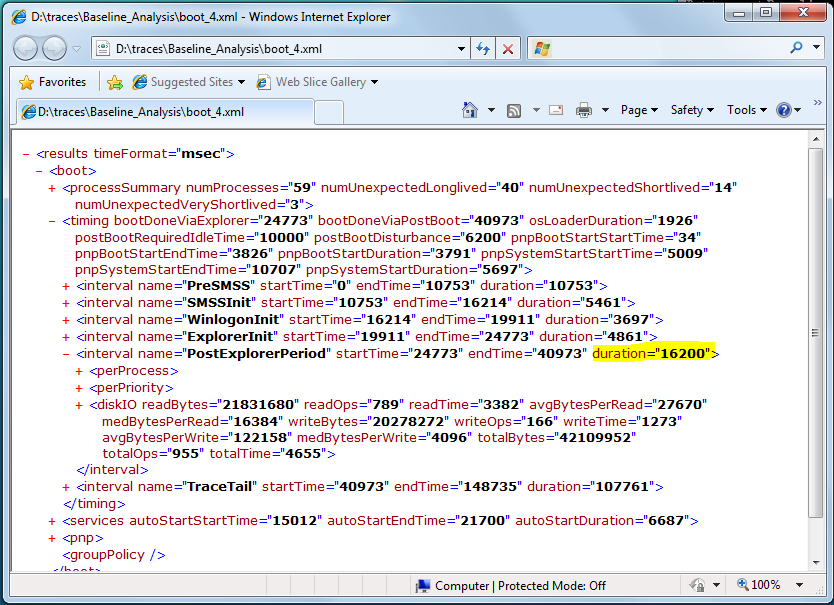


Figure 13. Location of PostBoot duration in Summary.xml

After the PostBoot phase begins, a user can start applications. However, in a controlled test environment for performance analysis, most of the PostBoot time typically involves completion of the loading and initialization of services and applications.

If your test system remains in PostBoot for an extended period, a poorly performing application or service might be preventing the system from becoming idle. Expand the **perProcess** and **diskIO** nodes and check whether a particular application or service uses an excessive amount of CPU or disk resources.

## Boot Transition: Summary

Users perceive boot time as an important aspect of system performance, so it is critical to optimize boot transition performance. The following is a list of important things to consider:

* Use WPT to measure the performance of drivers, applications, and services during all on/off transitions.
* Use Windows Driver Foundation (WDF) to write drivers.
* Minimize BIOS initialization time by configuring the system BIOS for fast POST.
* Ensure that all BOOT\_START drivers are embedded-signed to avoid the performance cost of checking catalog files.
* Mark drivers as BOOT\_START only when it is required.
* Increase parallelism during device enumeration and startup by returning STATUS\_PENDING for IRP\_MN\_START\_DEVICE and IRP\_MN\_QUERY\_DEVICE\_RELATIONS requests.
* Ensure that all nonessential services are converted to demand start or trigger start to make system resources available during boot.
* Avoid using load order groups to express service dependencies.
* Reduce application resource consumption during PostBoot.
* Limit background processing after boot by minimizing the number of applications that start from the **Run** or **RunOnce** registry keys.
* Perform testing in a controlled way, and make comparisons against a valid baseline. Obtain a baseline measurement on a system with as few system extensions as possible. Add devices, applications, and services one at a time, and test for unacceptable regressions in on/off transition times.

For details about these recommendations, see “Windows On/Off Transitions Solutions Guide” on the WHDC Web site.

# Sleep and Hibernate Transitions

Sleep and hibernate are key Windows customer scenarios. Improving performance of these transitions increases convenience and improves battery life for users. If a sleep or hibernate transition takes too long, users may choose to shut down their computers instead and therefore lose key advantages. For instance, Wake-on-LAN, an extremely useful feature for conserving energy and providing remote access to a computer, is possible only if the system is in the sleep or hibernate state. However, sleep and hibernate transitions have many potential vulnerabilities and challenges.

During the sleep transition, the system enters a low power state and retains context in main memory.

During the hibernate transition, system context is saved on disk and the system is completely powered down. However, systems with large amounts of RAM might face performance challenges because more information must be stored on disk during suspend and restored into memory on resume.

**Note**. The terms “sleep,” “suspend,” and “standby” are sometimes used interchangeably. In this paper, “sleep” includes two phases: suspend and resume. In the suspend phase, the system enters a low-power state and in the resume phase, the system returns to the operational state. In earlier versions of Windows, “standby” was sometimes used to describe a low-power state in which in the system was not completely shut down.

Sleep and hibernate transitions are divided into two high-level phases, namely Suspend and Resume. These are further divided into many subphases, each of which is vulnerable to specific performance problems caused by software inefficiencies, bugs, and configuration issues.

The PostResume subphase is particularly important for performance. Performing minimal activity during this subphase is critical to providing a high-quality experience when users resume their systems. For more information, see “PostResume Subphase” later in this paper.

This section provides an overview of the sleep and hibernate transitions and describes how to generate and analyze traces.

## Overview of the Suspend Phase

The subphases of the Suspend phase for hibernate and sleep transitions are shown in Figure 14.



Figure 14. Subphases of the Suspend phase for the sleep and hibernate transitions

## Overview of the Resume Phase

The subphases of the Resume phase for the sleep and hibernate transitions are shown in Figures 15 and 16, respectively.

**Figure 15. Subphases of the Resume phase of the sleep transition**

Figure 16. Subphases of the Resume phase of the hibernate transition

## Sleep and Hibernate Transitions: Capturing Traces

You can use Xbootmgr to capture traces of sleep and hibernate transitions. In the following example, the system transitions to sleep three times and the trace is saved to the traces directory on the root drive:

xbootmgr –trace standby –numRuns 3 –resultPath %systemdrive%\traces –postBootDelay 120 -traceFlags base

To capture a trace of the hibernate transition, replace **-trace standby** with **‑trace hibernate** in the preceding example.

The following are some key points to remember:

* The Xbootmgr tool does not reboot the test system before the standby tests run.
* The **-postBootDelay** value specifies the time-out in seconds that the system waits after resuming before stopping the trace. The default is 120, which is usually acceptable for suspend/resume testing.
* The **-traceFlags** **base** option provides a more lightweight and consistent trace from a timing perspective.
* A status dialog box appears during each iteration.
* After testing is complete, Xbootmgr creates the final trace but does not display a dialog box.

## Sleep and Hibernate Transitions: Analyzing Traces

You can analyze sleep and hibernate transition traces by using the Xperf.exe and Xperfview.exe tools in the WPT. Xperf can create Summary.xml files for simplified analysis, and Xperfview displays trace data in graphical form.

To generate an XML summary of the Suspend and Resume phases for either transition, use the **-a suspend** action with Xperf, as shown in the following example:

xperf -i trace.etl -o summary.xml -a suspend

Open the Summary.xml file in Internet Explorer. To enable dynamic node expansion, you might need to click the gold Information bar and then click **Allow blocked content**. You should see XML output that resembles that in Figure 17.

In the report, most intervals and operations are described by start time and duration.

Many phases involve substantial processing by applications, services, and devices. Detailed timing information appears in the subnode that is associated with each **scenario** node.

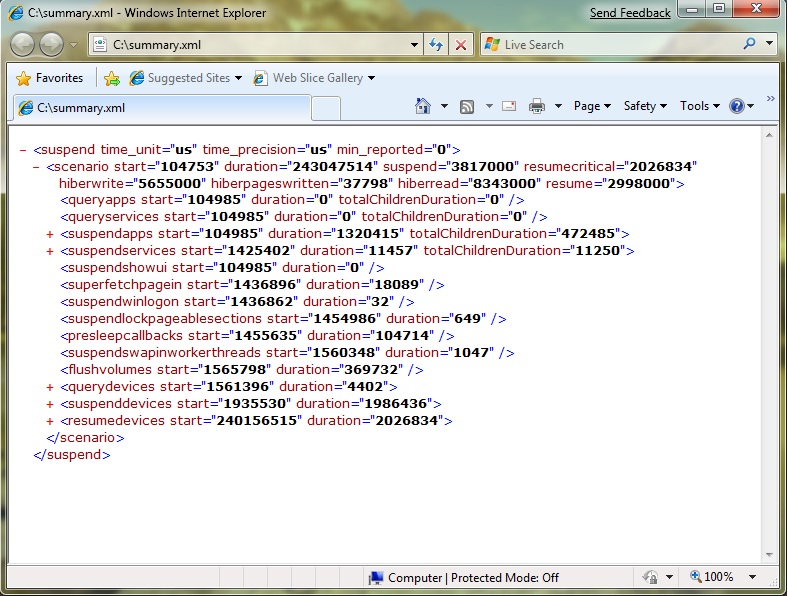


Figure 17. The Summary.xml file generated by using the suspend action in Xperf

The following notes apply to the Summary.xml file:

* The **scenario node** containshigh-level summary information.
* **duration** is the total time for the both the Suspend and Resume phases.
* **suspend** is the time that is required to suspend the system.
* **biosinit** is the time that is spent in the BIOs to resume the system. This metric is available only for the sleep transition, not for the hibernate transition.
* **resumecritical** is the time that is spent performing actions that are critical to resuming the system, such as resuming devices.
* **resume** is the time that the thread that controls the resume transition requires to finish executing the resume work (sending notifications and so on).
* Hibernate and resume transitions have **hiberwrite** and **hiberread** attributes.

Currently no attribute corresponds to the PostResumesubphase.

Hybrid sleep, which was introduced in Windows Vista, combines sleep and hibernate. The system enters sleep after it saves a minimal system context to disk. This allows the system to be restored even if an unexpected power failure occurs. Such transitions have the **hiberwrite** but not the **hiberread** attribute.

## Sleep and Hibernate Transitions: SuspendApps Subphase

### What Happens in SuspendApps

The SuspendApps subphase corresponds to processing the suspend message by all windowed applications. During SuspendApps, Windows serially sends a WM\_POWERBROADCAST message with an event type of PBT\_APMSUSPEND to all applications. This message notifies each application that the system is preparing to suspend and gives it a maximum of 2 seconds to process the message. If the application does not complete all its work before this time-out, any remaining processing completes after the system resumes from the sleep state.

### SuspendApps Performance Vulnerabilities

The SuspendApps subphase is vulnerable to performance issues if applications delay processing of the WM\_BROADCAST message.

Most applications are not required to do significant work during the Suspend phase. Any such work directly causes a delay in suspending the system because the notification process is serial. If several other processes are running concurrently on the system and consuming CPU resources, an application might not have sufficient CPU time to complete its work before the 2‑second time-out elapses.

### SuspendApps Performance Analysis

After you capture the traces and generate the Summary.xml file as described earlier in this paper, open the Summary.xml file in an XML reader such as Internet Explorer. Detailed per-process timing for the SuspendApps subphase appears in the **suspendapps** node.

The **suspendapps** node has the following three attributes:

* **start** identifies the beginning of this subphase.
* **duration** is the entire duration of this subphase.
* **totalChildrenDuration** is the sum of durations of all applications under this node.

The applications are listed in descending order of individual duration. For each application that is listed under the **suspendapps** node:

* **start** is the time when the application receives the suspend notification.
* **duration** is the total time that the application takes to process the notification.
* **exe** provides the path and name of the application.
* **pid** is the process ID.

In the Summary.xml file, the **duration** and **totalChildrenDuration** attributes under the **suspendapps** node have different values. The difference in these values (usually 1 second or less) is the time that the video driver requires to fade out the display and turn off the monitor. In Figure 17, this value is 1320415 – 472485 = 847930µs (or approximately 847 ms).

## Sleep and Hibernate Transitions: SuspendServices Subphase

### What Happens in SuspendServices

The SuspendServices subphase corresponds to processing the suspend notification by services. Only services that have opted to receive power notifications receive this notification, which is serially sent to each service.

### SuspendServices Performance Vulnerabilities

Most services are not required to complete significant work during this subphase. However, any work that occurs uses hardware resources, which can delay the system from entering the sleep state.

Services that are not required to complete any critical work during the Suspend phase can choose not to receive suspend notifications.

### SuspendServices Performance Analysis

The **suspendservices** node in the Summary.xml output file includes detailed per-process timing for this subphase. All services that opted to receive suspend notifications appear in this node, as Figure 18 shows.

Similar to the **suspendapps** node, the **suspendservices** node contains three attributes: **start**, **duration** and **totalChildrenDuration**. Each service has a start time, duration, and name under the **service** subnode.

The duration of this subphase is usually small because services typically have little or no work to do.

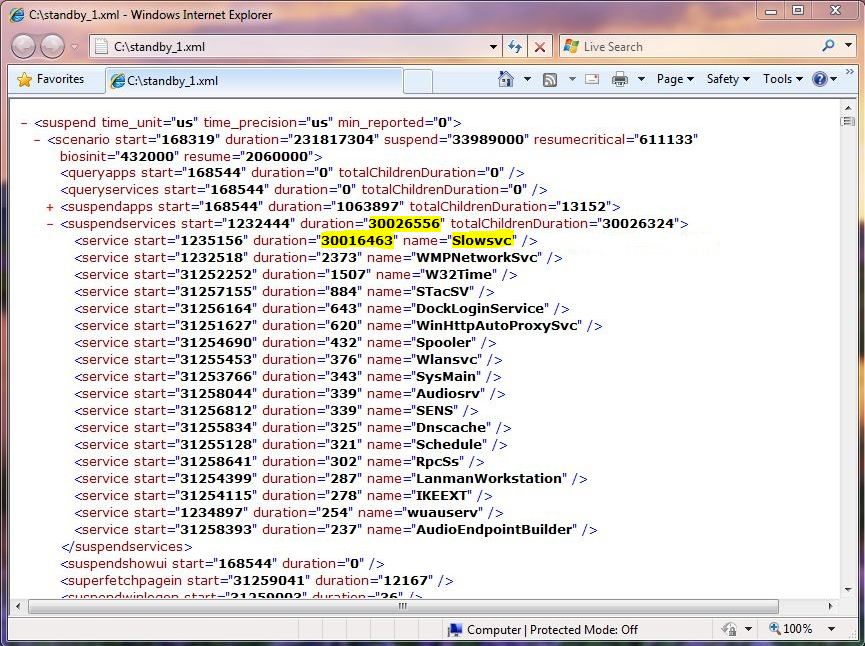


Figure 18. The suspendservices node in Summary.xml

In some cases, services can significantly delay sleep by performing network-related activities, writing lots of data to disk, or failing to respond quickly to suspend notifications.

In Figure 18, the service that is named Slowsvc delays system sleep for about 30 seconds.

To investigate a slow service

1. Note the corresponding **start** timestamp in the **service** subnode.

2. Open the trace in Xperfview.

3. In the Services graph, find the container process that hosts the service.

4. In the CPU Scheduling graph, highlight the region of interest, right-click it, and then choose **Summary Table with Ready Thread**.

5. Look for the container process in the summary table. The corresponding wait and ready stacks show which stack caused the delay and which stack cleared the wait.

In this example, we found that this service does not respond to the suspend notification from the SCM and therefore the SCM times out after 30 seconds.

## Sleep and Hibernate Transitions: QueryDevices and SuspendDevices Subphases

### What Happens in QueryDevices and SuspendDevices

Devices and drivers can significantly affect overall system sleep times. In the QueryDevices subphase, the power manager sends an IRP\_MN\_QUERY\_POWER request for the system sleep or hibernation power state so that devices can prepare for the power state change. The function driver for each device receives this system IRP and sends a device power query IRP down the device stack.

After all devices have been queried, the system enters the SuspendDevices subphase. Here, the power manager sends an IRP\_MN\_SET\_POWER request for a lower system power state. Device drivers must then save appropriate device context and prepare the device for the system to enter sleep or hibernate.

Drivers can delay the suspend transition in both the QueryDevices and SuspendDevices subphases.

To identify performance problems in these subphases, compare the results against a baseline measurement.

The Summary.xml output file provides per-driver and per-device summaries in the **queryDevices** and **suspendDevices** nodes as shown in Figures 19 and 20.

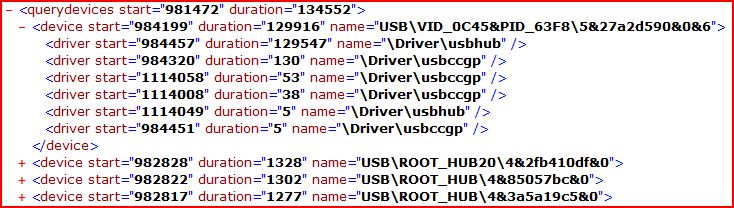


Figure 19. A snippet of the querydevices node from Summary.xml

The Summary.xml file contains the start and total duration of the QueryDevices subphase as attributes of the **querydevices** node. Under this node, each device is listed with its name, the start time, and the total time that was taken to process the query IRP. Figure 19 shows this information for a USB device. Figure 20 shows the **suspenddevices** node.

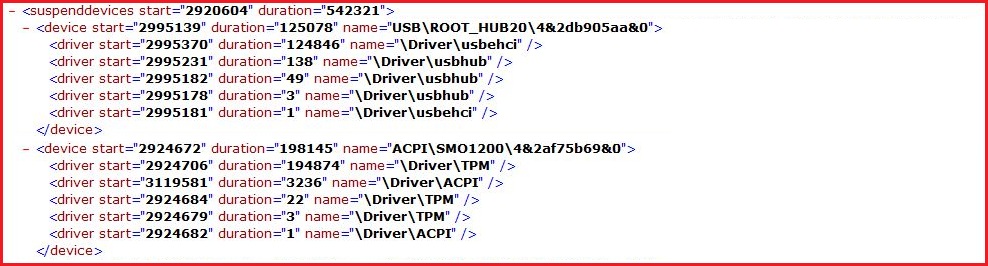


Figure 20. A snippet of the suspenddevices node from Summary.xml

The structure of the **suspenddevices** node is identical to that of the **querydevices** node. This node contains information about the time that is required to process the system set-power IRP.

## Sleep and Hibernate Transitions: ResumeDevices Subphase

### What Happens in ResumeDevices

Device and driver performance can have a large effect on the resume time of the system because all devices must resume before the system can restore the desktop. During the ResumeDevices subphase, the power manager notifies all drivers that the system is resuming by sending an IRP\_MN\_SET\_POWER request for S0 to all devices. This subphase is considered complete when all drivers have completed the S0 set-power IRP.

### ResumeDevices Performance Vulnerabilities

Drivers that do not implement fast resume can cause serialization delays and reduce overall system resume performance. Drivers that use WDF do not require any special code because WDF automatically implements fast resume. However, drivers that are built on the Windows Driver Model (WDM) must follow the steps in “Windows On/Off Solutions Transitions Guide” on the WHDC Web site.

Drivers that use excessive CPU resources can also cause delays on resume from sleep or hibernate. Excessively long deferred procedure call (DPC) functions can block lower priority threads from executing or even block other DPC functions that are queued and ready to execute. This increases the duration of the device initialization phase and extends the overall system resume time.

### ResumeDevices Performance Analysis

As with the QueryDevices and SuspendDevices subphases, you should validate the ResumeDevices data against the correct baseline after changes to drivers and devices. The Summary.xml output file provides per-driver and device summaries in the **resumedevices** node, as shown in Figure 21.

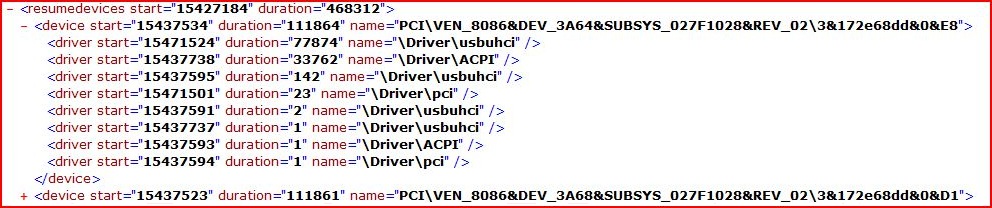


Figure 21. A snippet of the resumedevices node from Summary.xml

The **resumedevices** node has start time and total duration as attributes and contains per-device and driver summaries as subnodes.

## Sleep and Hibernate Transitions: ResumeServices and ResumeApps Subphases

### What Happens in ResumeServices and ResumeApps

The ResumeApps and ResumeServices subphases begin immediately after the ResumeDevices subphase ends. During the ResumeApps subphase, the kernel power manager calls into the USER subsystem and requests that it send a WM\_POWERBROADCAST window message to all applications. For services, it calls a private interface in the user-mode PnP manager, which then calls each service’s power management event handler.

The ResumeServices and ResumeApps subphases are asynchronous, and the system considers them complete after it notifies the infrastructure that manages the resume process to resume.

These two subphases are a common source of performance problems. For example, consider how much time passed while the system was suspended. After the system resumes, many timers and scheduled tasks can start and many applications and services can receive concurrent notifications. The resulting activity often leads to high CPU and disk usage, which causes contention between user activity and background resume activity.

For information about how to reduce performance problems that occur during these subphases, see “Windows On/Off Transitions Solutions Guide” on the WHDC Web site.

## Sleep and Hibernate Transitions: PostResume Subphase

### What Happens in PostResume

The PostResumesubphase is a conceptual subphase that covers the time that follows the resume transition. As with the boot transition and the PostBoot subphase, the goal is to get the system to an idle state as quickly as possible.

The CPU Sampling by CPU and Disk Utilization graphs in Xperfview can provide insight into how busy the resumed system is, but a valid baseline measurement is important when you analyze PostResume activity. Currently, no tools support the calculation of a PostResume metric. Instead, we strongly recommend that you compare traces from the baseline and the test system to identify anomalous CPU and disk activity that occurs during the PostResume period.

## Sleep and Hibernate Transitions: Summary

The phases of the sleep/hibernate transitions are complex but important phases to optimize. The following is a list of important things to consider:

* Use the WPT to measure the performance of drivers, applications, and services during all on/off transitions.
* Use WDF to write drivers.
* Ensure that all drivers implement fast resume and optimize CPU resource usage on resume to achieve better resume performance.
* Verify that services do not opt to receive suspend notifications unless absolutely required.
* Ensure that all applications and services respond quickly to resume events and minimize post-resume CPU, disk, and network usage. Performance in the PostResume subphase is absolutely critical to customer satisfaction.
* Avoid delays in processing the suspend notification (WM\_POWERBROADCAST message) in applications.
* Optimize resume from hibernate performance by improving BIOS INT13 performance.
* Perform testing in a controlled way, and make comparisons against a valid baseline. Obtain a baseline measurement on a system with as few system extensions as possible. Add devices, applications, and services one at a time, and test for unacceptable regressions in on/off transition times.

For details about these recommendations, see “Windows On/Off Transitions Solutions Guide” on the WHDC Web site.

# Shutdown Transition

Fast shutdown is important for a successful overall user experience on a system. A lengthy shutdown time can cause frustration among users, increase service times for system administrators, and adversely affect the reliability of systems. For example, if shutdown takes too long, the user might press the power button instead and could potentially lose data.

Services and applications can have a significant effect on shutdown performance. CPU and disk activity during the shutdown process can also delay the shutdown transition.

Figure 22 shows the three main phases of the shutdown transition.



Figure 22. Phases of the shutdown transition

## Shutdown Transition: Capturing Traces

You can use Xbootmgr to capture a trace of the shutdown transition. In the following example, Xbootmgr restarts the system three times and saves the trace in the Traces directory on the root drive:

xbootmgr -trace shutdown -numRuns 3 -resultPath %systemdrive%\traces   
-postBootDelay 180 -traceFlags base

The following are some key points to remember:

* The **–postBootDelay** value specifies the time-out in seconds that the system waits after booting before it starts the next shutdown trace. The default is 120, but we recommend 180 seconds for shutdown traces.
* The **-traceFlags** **base** option provides a lightweight and consistent trace.
* Xbootmgr reboots the system one time before it collects the first trace.
* Xbootmgr reboots the system for each iteration, and a status dialog box appears throughout each iteration.
* After testing is complete, Xbootmgr reboots the system another time, but no status dialog box appears.

## Shutdown Transition: Analyzing Traces

You can use the Xperf.exe and Xperfview.exe tools in the WPT to analyze the shutdown transition performance data.

To generate an XML summary of shutdown, use the **-a shutdown** action with Xperf, as shown in the following example:

xperf -i trace.etl -o summary.xml -a shutdown

Open the Summary.xml file in an XML reader such as Internet Explorer. The report contains detailed per-phase timing information as Figure 23 shows.

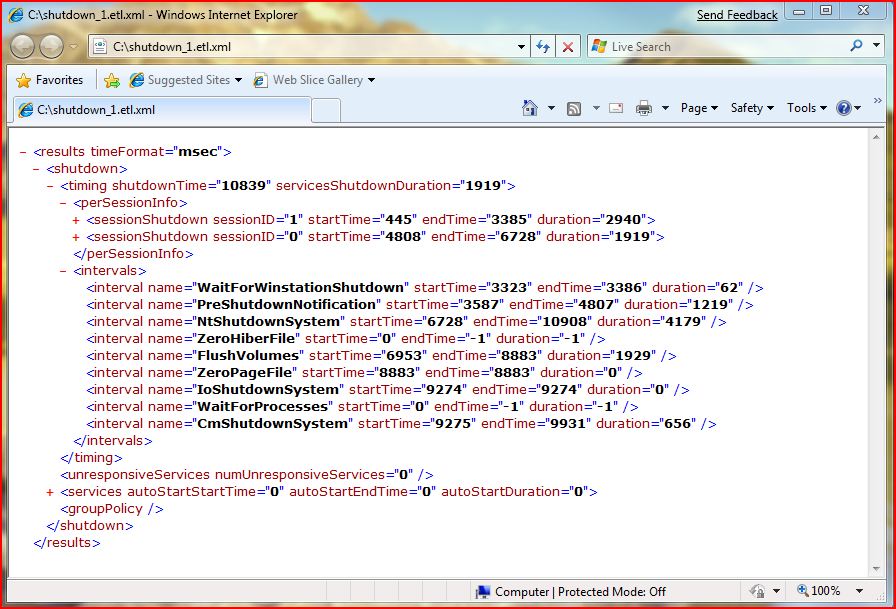


Figure 23. Sample Summary.xml file generated in Xperf with the shutdown action

The following are some key points to remember:

* The time unit that is used (such as milliseconds) appears at the top of the report.
* The **timing** node specifies the total time for the system to shut down and for all services to shut down.
* The **perSessionInfo** node contains the start time, end time, and duration for all applications that have session ID 0 or 1.
* The **intervals** node lists all activities that occur between the shutdown phases and the start time, end time, and duration for each activity.
* The **services** node lists all services and the start time, end time, and time to shut down each service.

## Shutdown Transition: UserSession Phase

### What Happens in UserSession

During this phase, the Client/Server Runtime Server Subsystem (Csrss.exe) shuts down all applications that are running in the user session—that is, all applications that have session ID 1.

When the user chooses **Shut Down** from the **Start** menu, Csrss sends two shutdown notifications (WM\_QUERYENDSESSION and WM\_ENDSESSION messages) to each user interface (UI) thread in each graphical user interface (GUI) application. If after 5 seconds any application blocks shut down, Windows displays the dialog box in Figure 24 so that users can choose to force or cancel shutdown.

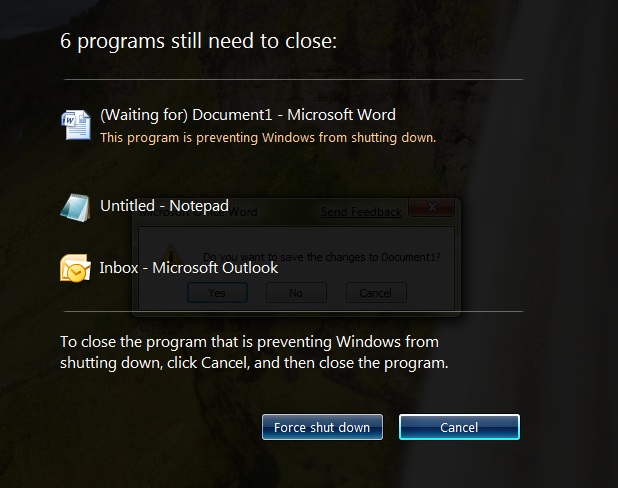


Figure 24. Shutdown UI in Windows 7 and Windows Vista waiting for user input

To shut down a console application, Windows invokes the console control handler by sending CTRL\_LOGOFF\_EVENT and waits up to 5 seconds for a response.

If the user selects **Force shut down** from the dialog box, the system gives all applications 0.25 second to respond to WM\_QUERYENDSESSION messages and 0.5 second to respond to WM\_ENDSESSION messages, and then closes any remaining applications.

For system-forced shutdowns—which a user starts by running Shutdown.exe or an application starts by calling the corresponding system function—Windows does not display a dialog box. Instead, it immediately closes any hung applications and closes all GUI and console applications after the 5‑second time-out. For example, shutdowns that are triggered by Windows Update are considered system-forced shutdowns.

### UserSession Performance Vulnerabilities

Applications must perform a handshake with Csrss, and any applications that do not do this correctly are timed out and closed.

Because Windows serially shuts down applications, any delay in a process’s shutdown path contributes to the total shutdown duration. To ensure a speedy shutdown, every application must respond quickly to shutdown notification messages (WM\_QUERYENDSESSION and WM\_ENDSESSION).

Windows uses long time-outs so that applications have sufficient time to shut down and save user data. Therefore, applications can have a significant effect on shutdown performance.

### UserSession Performance Analysis

To analyze the performance of this phase, you can use the Summary.xml output file that you created previously in “Shutdown Transition: Analyzing Traces.”

The time that is required to shut down an application varies among applications. Application shutdown timings appear in the **sessionShutdown** subnode of the **perSessionInfo** node as Figure 25 shows.

The sessionID attribute in this phase is 1, which indicates the user session.

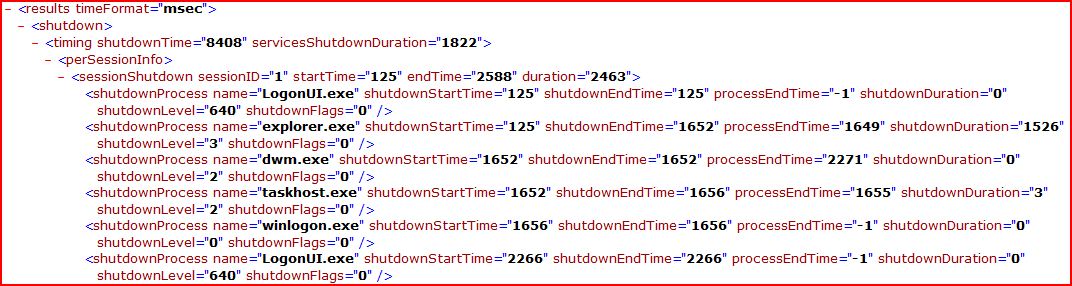


Figure 25. Snippet of the Summary.xml file that shows application shutdown times

To identify the cause of any process that delays shutdown, note the **shutdownStartTime** and **shutdownEndTime** for that process in the **shutdownProcess** subnode and open the trace in Xperfview. The CPU Sampling by Process graph in Xperfview can help you identify CPU consumption issues. The CPU Scheduling graph shows wait and ready stacks for each thread in the process and can help you identify delay issues.

For specific instructions on how to identify applications that delay system shutdown, see Appendix A.

Some applications can take a long time to process the shutdown notification and thereby delay other applications from receiving notifications. Application developers should follow these guidelines to reduce delays in the shutdown path:

* Avoid unnecessary disk flushes at the time of shutdown by minimizing the amount of unsaved data at shutdown. To do so, perform regular disk flushes during processing (that is, regularly and automatically save data), keep track of data that is saved to disk, and save only unsaved data at shutdown. In addition, avoid explicit flushes such as the following two examples:

Disk flushes that the application initiates, such as calling **FlushFileBuffers**.

Registry flush on shutdown. Avoid calling explicit flushes during shutdown because this increases overall shutdown time.

* Avoid unnecessary network dependencies during shutdown. Network problems can delay the shutdown of applications.

## Shutdown Transition: SystemSession Phase

### What Happens in SystemSession

This phase includes two subphases:

* **Preshutdown notification.** Windows serially shuts down all services that registered to receive preshutdown notifications. Ordered services—services that have set up the shutdown order of dependent services—are shut down before non-ordered services.
* **Shutdown notification.** All services that registered to receive shutdown notifications are shut down in parallel.

If all services have not exited after 20 seconds (in Windows Vista) or 12 seconds (in Windows 7), the system continues the shutdown. Processes and services that do not shut down in a timely manner are left running as the system shuts down.

### SystemSession Performance Vulnerabilities

Only services that must handle preshutdown and shutdown notifications should opt to receive them. In the preshutdown notification subphase, the SCM serializes the waits. Therefore, these services block system shutdown until they exit or until the *wait hint* time-out expires.

Services are not guaranteed to have enough time to finish all their work in the shutdown notification subphase before the system shuts down.

If a service registers to receive shutdown notifications through its service control handler, it must respond quickly to notifications from the SCM. To correctly shut down without adversely affecting shutdown transition time, a service should:

* Mark itself as STOP\_PENDING immediately after it receives the shutdown notification. This returns control to the SCM.
* Enter the stopped state (SERVICE\_STOP) as soon as possible.

For specific guidelines on how to write service control handler functions, see “Service Control Handler Function” on MSDN.

### SystemSession Performance Analysis

By looking at the Summary.xml output file that you generated in “Shutdown Transition: Analyzing Traces,” you can see the total time for this phase in the **servicesShutdownDuration** attribute of the **timing** node. This value appears in the third line in Figure 26.



Figure 26. Snippet of the Summary.xml file that shows service shutdown times

Services that do not perform the SCM handshake correctly and therefore delay shutdown are identified in the **unresponsiveServices** node. Service developers must ensure that their services respond quickly to notifications from SCM because unresponsive services can delay shutdown by 12 seconds (in Windows 7) or 20 seconds (in Windows Vista). The sample output in Figure 26 shows no unresponsive services.

Following the **unresponsiveServices** node, the **services** node lists individual service names, start times, end times, and total durations. The start time indicates the time at which the service began to process the shutdown notification.

You can also use the Xperfview tool to see services that delay shutdown. Open the trace file by using Xperfview.exe, and then check the **Services** box in the **Frame List** on the left, as shown in Figure 27.

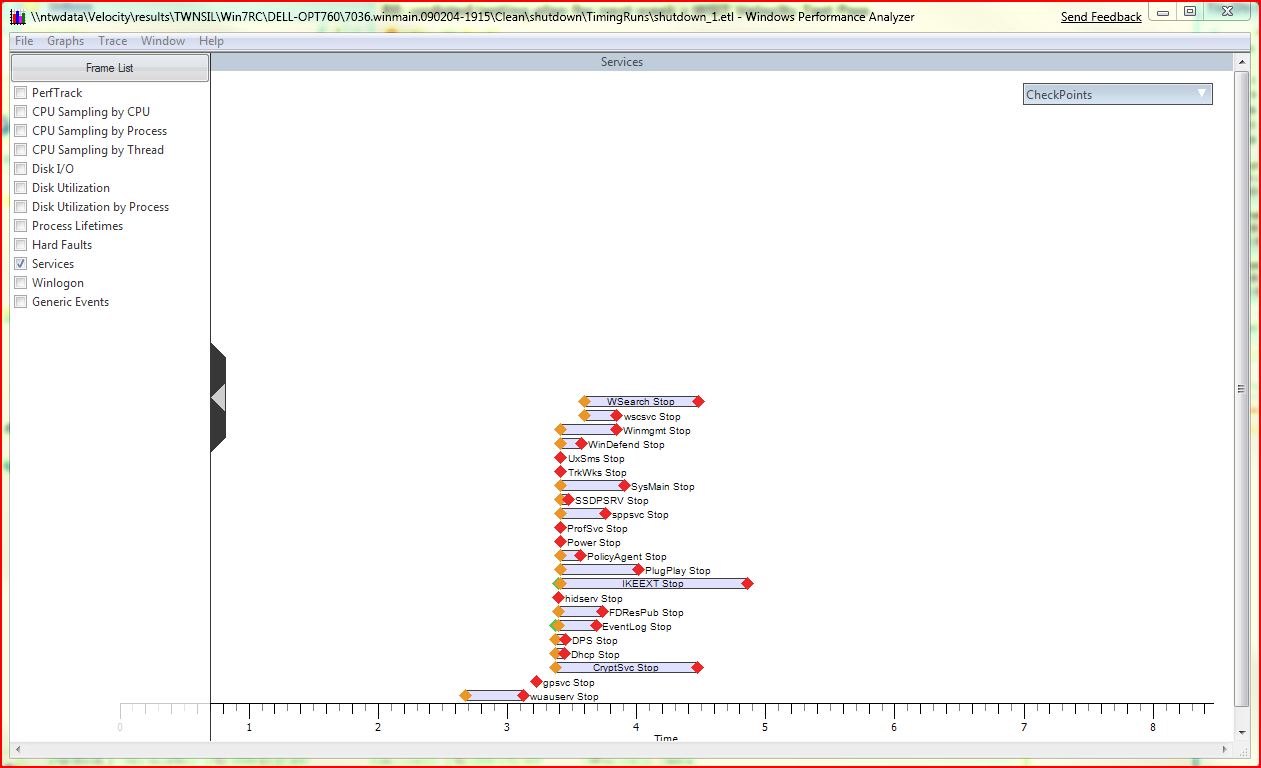


Figure 27. A sample trace file in Xperfview that shows service shutdown times

## Shutdown Transition: KernelShutdown Phase

### What Happens in KernelShutdown

In the KernelShutdown phase, the rest of the system, including all devices and drivers, is shut down.

To calculate the time spent in KernelShutdown

1. In the XML trace file, locate the overall shutdown time. For example:

<**results** **timeFormat**="msec">

<**shutdown**>

<**timing shutdownTime**="13066" **servicesShutdownDuration**="2675">

<**perSessionInfo**>

<**sessionShutdown** **sessionID**=”1” **startTime**=”448” **endTime**=”7091”

**duration**=”6642”>

<**sessionShutdown** **sessionID**=”0” **startTime**=”8335” **endTime**=”11010”

**duration**=”2675”>

2. Subtract the time that is required to shut down the system and user sessions from **shutdownTime**. In this example, the KernelShutdown phase is calculated as 13066 – 6642 – 2675 = 3,749 milliseconds.

## Shutdown Transition: Summary

The following is a list of important considerations to help you improve shutdown performance. For details about these recommendations, see “Windows On/Off Transitions Solutions Guide” on the WHDC Web site.

* Use the WPT to measure the performance of drivers, applications, and services during all on/off transitions.
* Ensure that services do not opt to receive preshutdown and shutdown notifications (SERVICE\_CONTROL\_PRESHUTDOWN and SERVICE\_CONTROL\_SHUTDOWN control codes) unless they are absolutely required.
* Ensure that all services that opt to receive shutdown notifications respond quickly to the SCM.
* Ensure that all applications respond quickly to shutdown notifications (WM\_QUERYENDSESSION and WM\_ENDSESSION messages).
* Reduce delays in the shutdown path of services and applications by minimizing CPU, disk, and network activity in response to shutdown notifications.
* Use the **perSessionInfo** and **sessionShutdown** nodes to see timing for each application.
* Perform testing in a controlled way, and make comparisons against a valid baseline. Obtain a baseline measurement on a system that has as few system extensions as possible. Add devices, applications, and services one at a time, and test for unacceptable regressions in on/off transition times.

# Conclusions

You can identify and solve most on/off transition performance problems by using the tools and techniques in this paper. For example, system integrators can incorporate on/off testing by using Xbootmgr.exe, Xperf.exe, and Xperfview during the preinstallation image validation procedure. Such testing can provide significant value to users.

Furthermore, by using these tools during hardware and application development, you can discover hardware and software issues that might otherwise go unnoticed.

# Resources

#### White Papers on the WHDC Web Site

Windows On/Off Transitions Solutions Guide

<http://msdn.microsoft.com/en-us/windows/hardware/gg463230.aspx>

Performance Testing Guide for Windows

<http://msdn.microsoft.com/en-us/windows/hardware/gg463399.aspx>

Mobile Battery Solutions Guide for Windows Vista

<http://msdn.microsoft.com/en-us/windows/hardware/gg487544.aspx>

Mobile Battery Life Solutions for Windows 7: A Guide for Portable Platform Professionals

<http://msdn.microsoft.com/en-us/windows/hardware/gg487547.aspx>

Application Power Management Best Practices for Windows Vista

<http://msdn.microsoft.com/en-us/windows/hardware/gg463247>

Handling IRPs: What Every Driver Writer Needs to Know

<http://msdn.microsoft.com/en-us/windows/hardware/gg487398.aspx>

Developing Efficient Background Processes for Windows

<http://msdn.microsoft.com/en-us/windows/hardware/gg463211.aspx>

#### Documentation on MSDN

QUERY\_SERVICE\_CONFIG Structure

<http://msdn.microsoft.com/en-us/library/ms684950(VS.85).aspx>

SetProcessShutdownParameters Function

<http://msdn.microsoft.com/en-us/library/ms686227(VS.85).aspx>

On/Off Transition Trace Capture tool: Reference

<http://msdn.microsoft.com/en-us/library/ff191001(VS.85).aspx>

Windows Performance Toolkit

<http://msdn.microsoft.com/en-us/performance/cc825801.aspx>

Windows Performance Analyzer Command Line Reference

<http://msdn.microsoft.com/en-us/library/ff190873(VS.85).aspx>

Windows Performance Analyzer Symbol Support

<http://msdn.microsoft.com/en-us/library/ff191023(VS.85).aspx>

#### Software Development Kits

Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1

<http://www.microsoft.com/downloads/details.aspx?FamilyID=c17ba869-9671-4330-a63e-1fd44e0e2505&displaylang=en>

Windows Driver Kit

<http://msdn.microsoft.com/en-us/windows/hardware/gg487428.aspx>

# Appendix A: Identifying On/Off Performance Issues by Using Xperf

This appendix contains several examples of potential performance issues that can occur during on/off transitions and provides specific instructions on how to use the WPT to identify these issues. The examples involve simulated applications, services, and drivers.

Each section describes a sequence of tasks that you should follow to diagnose issues in a particular phase.

## Boot: Identifying Boot and System Plug and Play Issues

#### Task 1: Observe device and driver start and initialization times in a baseline Summary.xml file

If you suspect performance issues with a driver that is loaded and initialized at boot, you can inspect the Plug and Play data for the driver. In your Summary.xml file, expand the **pnp** node. You should see three **phase** nodes under **pnp**, as Figure A‑1 shows. Note that the start time of the **bootStart** Plug and Play phase is 32 ms, which is near the beginning of the trace (time 0). By expanding the **bootStart** node, you can see all Plug and Play activity on a driver-by-driver basis, including the duration of each Plug and Play operation for each driver.

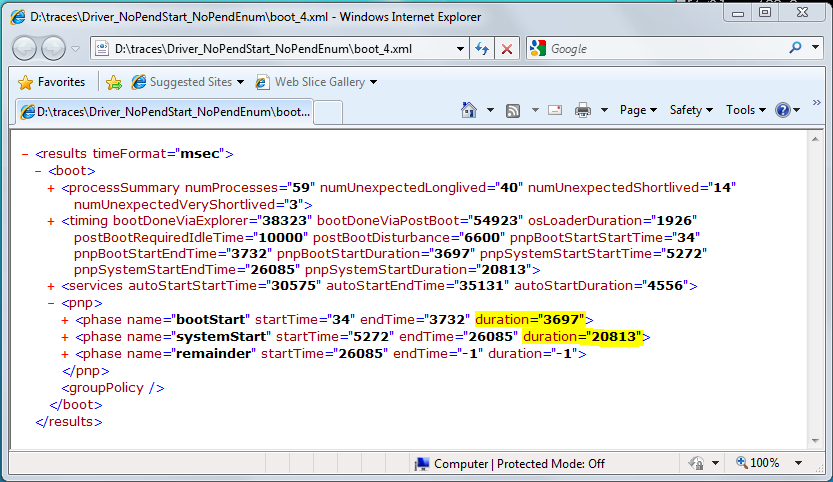


Figure A-1. The pnp node in a Summary.xml file

Another way to find Plug and Play data for BOOT\_START drivers is to generate a Summary.csv file, as described in Appendix C. To review the summary file, load the .csv file into Excel and sort the **PrePendInitTime** and **PostPendInitTime** columns from largest to smallest. The phase nodes that are named **bootStart** and **systemStart** under the **pnp** node contain Plug and Play performance data in XML format. You can expand these nodes to investigate Plug and Play performance, such as how pending the IRPs that are sent during device/driver initialization affects boot times. We created a sample driver that is named Ramdisk.sys and performed two test runs. In one test Ramdisk.sys returns STATUS\_PENDING for these IRPs, and in the other Ramdisk.sys completed the IRPs before it returned.

Figure A-2 shows the Boot.xml file for the poorly performing Ramdisk.sys, which does not pend the start or enumeration IRPs.

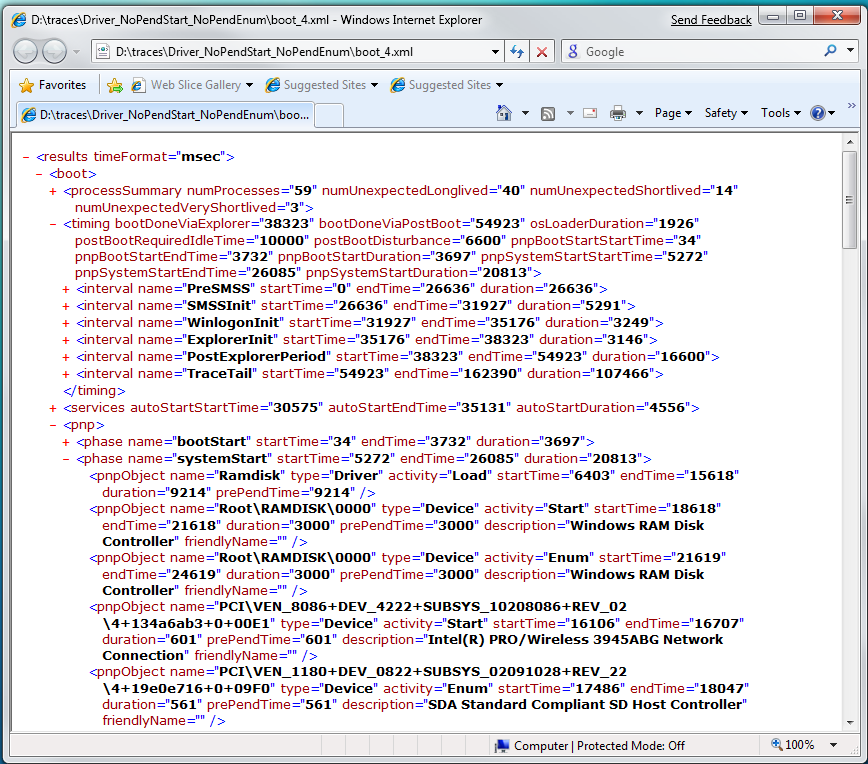


Figure A-2. Detailed Plug and Play data for poorly performing Ramdisk.sys

Note several data points in this XML output. Total boot time on this poorly performing system was 54.9 seconds. Approximately 26.6 seconds were spent in the PreSMSS phase, and the **systemStart** Plug and Play time duration was approximately 20.8 seconds. The first three **pnpObject** entries show that the duration of both the start and enumeration operations for Ramdisk was 3 seconds. The **PrePendTime** in each instance is equal to the duration. The **prePendTime** indicates how long the driver processed the IRP in its start or enumeration dispatch routine before it returned STATUS\_PENDING. This value should be as close to 0 as possible. If it equals the duration, the driver did not pend the IRP.

#### Task 2: Observe device and driver start and initialization times after system modification

For comparison, Figure A-3 shows the XML output from a system that uses a correctly written Ramdisk.sys, which returns STATUS\_PENDING for both operations.



Figure A-3. Detailed Plug and Play data for well-performing Ramdisk.sys

From this output, we see in the **timing** node that boot time has decreased to 45.4 seconds. **PreSMSS** is only 17.3 seconds, and the **systemStart** Plug and Play time is decreased to 12.3 seconds. Glancing further down in the output to the **pnpObject** entries, the start and enumeration durations for Ramdisk.sys were still 3 seconds, but the **prePendTime** is 0 for each item. Because the **pnpObject** entries appear in order of magnitude, we can presume that pending the IRPs eliminated the more than 9‑second delay that occurred in the poorly written Ramdisk.sys.

#### Task 3: Compare data from multiple runs

Table A-1 shows the data. For demonstration, we added data from a third test case in which we pended the start IRP but not the enumeration IRP.

Table A-1. Pending IRP Data for Ramdisk sample

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| System | PreSMSS time | Plug and Play system start time | Ramdisk start time | Ramdisk enumeration time | Total boot time | Boot time reduc­tion |
| No Pends | 26636 | 20813 | 3000 | 3000 | 54923 | Not applicable |
| Pend Start, not Enumeration | 17919 | 13007 | 3000 | 3000 | 46839 | 8084 |
| Pend Start and Enumeration | 17294 | 12343 | 3000 | 3000 | 45409 | 9514 |

Although the Ramdisk start and enumeration times in the table are the same for each test, the total boot time decreased in each run. Pending lets the PnP manager start other drivers in the device tree while the Ramdisk driver continues its own processing. This is a good example of how greater parallelism enables more use of system resources and therefore reduces boot times.

## Boot: Identifying Service Start Time Issues

#### Task 1: Observe service start times in the Summary.xml file

The **services** node in the Summary.xml file, which is highlighted in Figure A‑4, contains the performance data for individual services that start during boot.

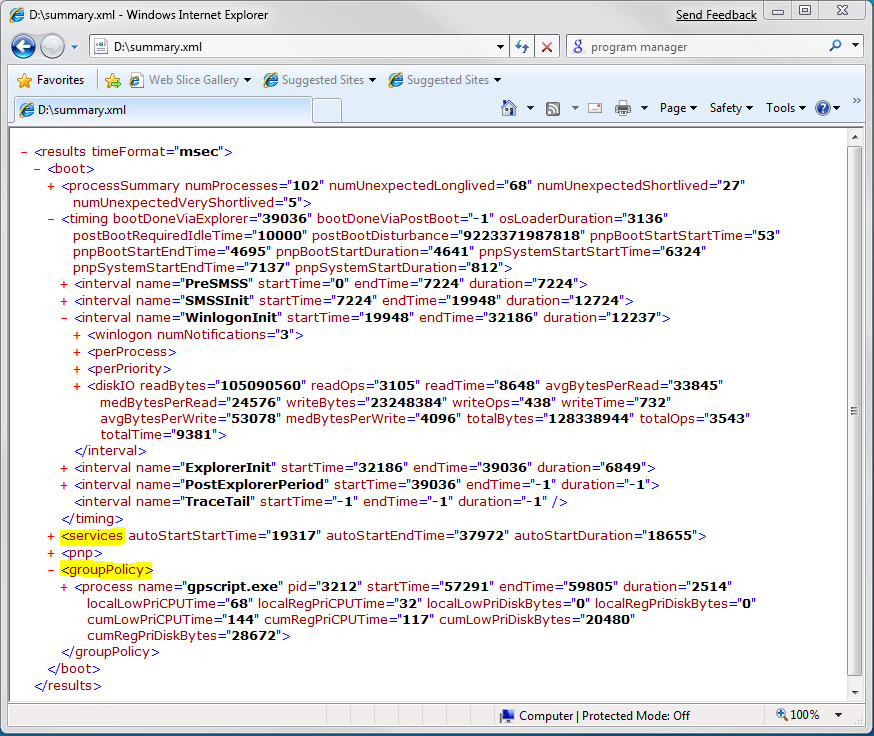


Figure A-4. The services node in the Summary.xml file

You can find service start timing information by examining the **services** node in the XML summary. Every service has its own subnode, and the time that is required for the service to begin appears in the **totalTransitionTimeDelta** attribute. Or, you can use the **xperf –a services** command or the Xperfview visualization tool to identify services that have long start delays.

This section discusses the Xperfview method to inspect a service that has a long start time and is assigned to an early load order group. The service is called Slowsvc, and its executable name is Delayservice.exe.

#### Task 2: Locate the service start time interval in the Services graph of Xperfview.exe

Open the trace by using Xperfview, which is a GUI-based tool that lets you visualize trace data by showing time along the X-axis. Figure A-5 shows the Services graph in Xperfview.

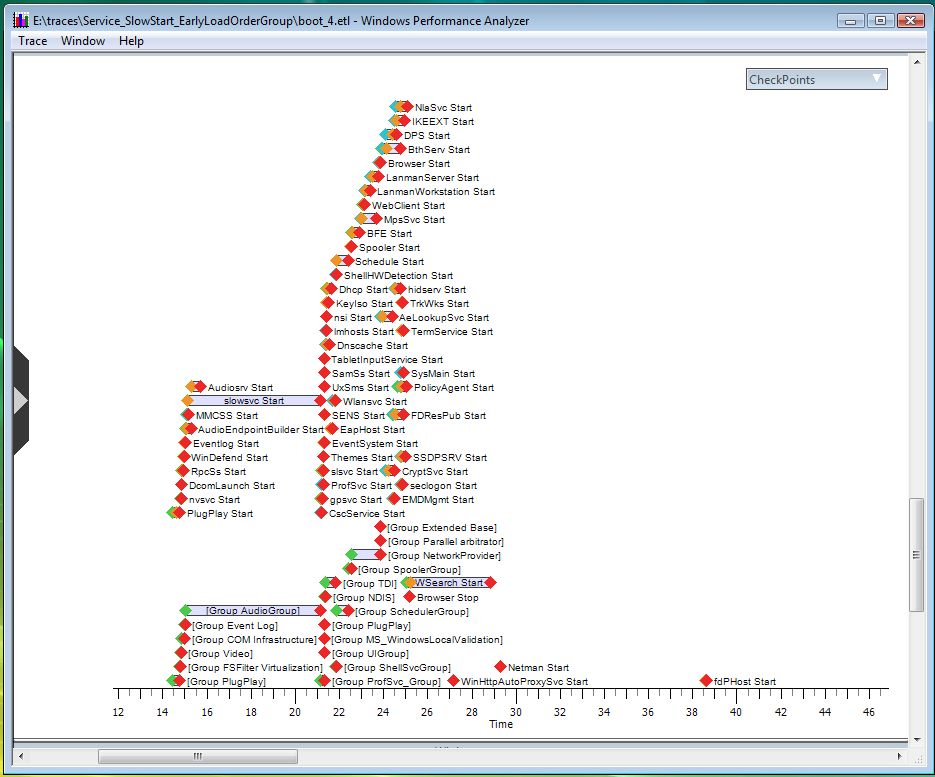


Figure A-5. Xperfview Services graph

In the figure, a horizontal bar represents the load time for each service.

Slowsvc clearly took longer than any other service on the system. We can infer from the length and overlap of the “Group AudioGroup” bar that slowsvc is part of the AudioGroup. Note that until slowsvc was finished loading (the end of the bar that is marked “slowsvc Start”), no other start event was logged on the system. Slowsvc Start ends at approximately 21 seconds, and all the dots that mark the starts for other services occur after the 21-second point. To verify this information in Xperfview, you can select areas of the chart, zoom in, and use the time axis at the bottom. An alternative method is to look at the XML or CSV output.

#### Task 3: Modify the service and locate it in Xperfview.exe

We prepared another trace in which the load order group does not include Slowsvc. In this trace, Slowsvc still has a 6-second initialization time, but the initialization time does not directly affect the critical boot path or total boot time.

Figure A-6 shows the Xperfview Services graph for the modified load order group.

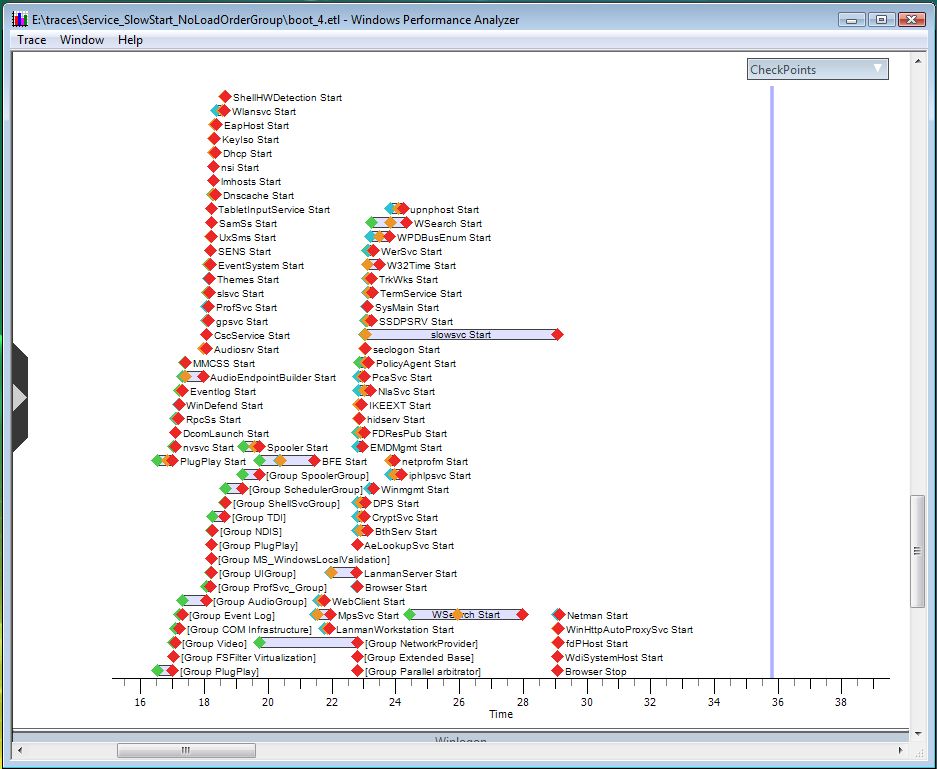


Figure A-6. Xperfview Services graph for modified load order group

As A-6 shows, Slowsvc start does not prevent other services from starting because it is not part of the load order group.

#### Task 4: Compare the modification to the baseline

Next, compare the boot time effects of the service in both situations by using the XML output, as shown in Figures A‑7 and A-8.

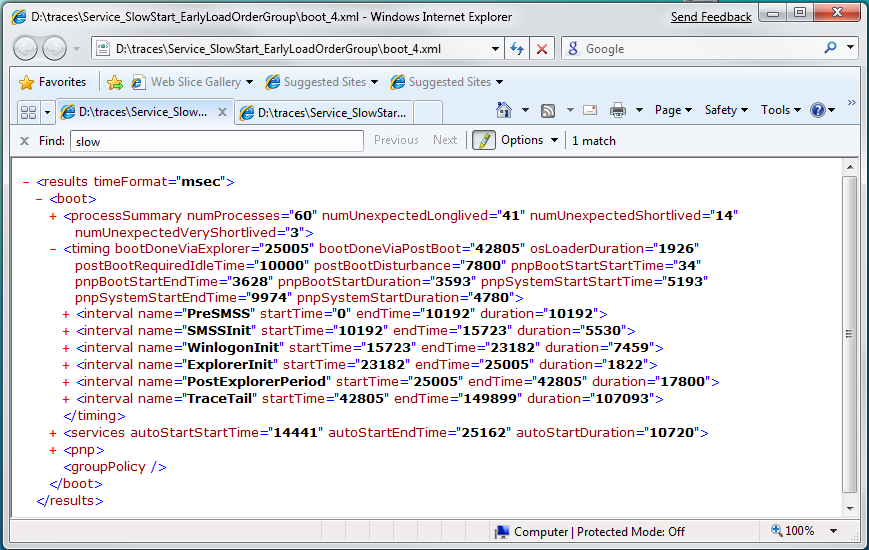


Figure A-7. XML output for Slowsvc in early load order group

Total boot time shown in Figure A-7 is 42.8 seconds, with 7.4 seconds spent in WinLogonInit and 1.8 seconds spent in ExplorerInit. Services required a total of 10.7 seconds to start.

Compare Figure A-7 to Figure A-8. After we removed the service from the load order group, total boot time decreased to 41.8 seconds.

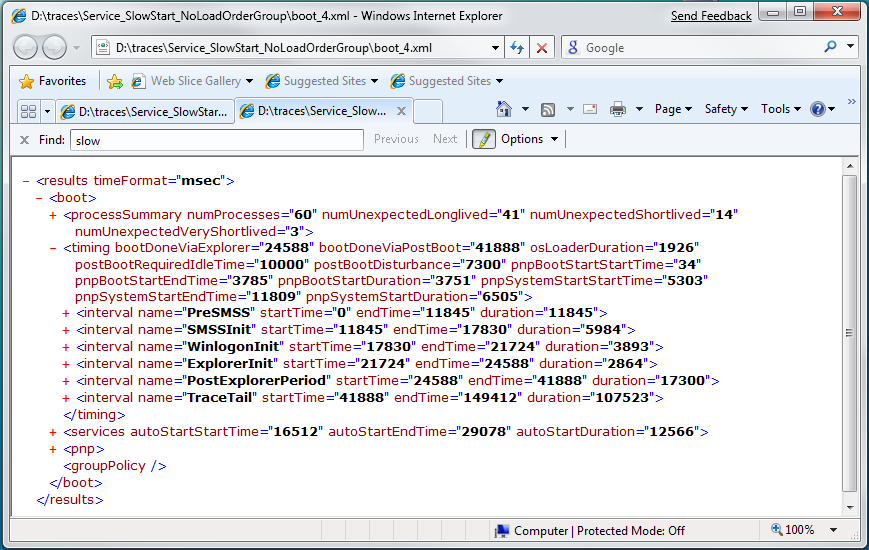


Figure A-8. XML output for Slowsvc removed from load order group

Table A-2 summarizes the data.

Table A-2. Differences in Boot Time with Slowsvc in Load Order Group

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| System | SlowsvcInit time | WinLogonInit time | | ExplorerInit time | Total services start time | Total boot time | Boot time reduc­tion |
| Load Order Group | 6000 | | 7459 | 1822 | 10720 | 42805 | Not applicable |
| No Load Order Group | 6000 | | 3893 | 2864 | 12566 | 41888 | 917 |

This data shows several concepts. First, services start throughout several boot phases, so looking at boot phase times is not a good way to evaluate service effects on boot. Because Slowsvc starts later in the boot transition when it is removed from the load order group, service start time is redistributed among the boot phases. This can cause the duration of a boot phase to increase, as the time for ExplorerInit did.

Second, and somewhat counter-intuitively, total service start time does not have a direct effect on total boot time. The service start time increased to 12.5 seconds in the second example, yet total boot time decreased. The reason is that service initialization runs in parallel with the system boot path after several key services and groups—those up to and including the ProfSvc Group—are running. The ProfSvc group is the final group that starts services that are required for logon.

Third and most important, not all boot optimizations create wall-clock reductions in the overall boot times. By increasing parallelization, as in this example, we eliminated a 6-second wait, yet gained only a 917-millisecond improvement in boot time. In this case, the system prefetched during the 6-second wait and the subsequent phases benefited from prefetching during WinLogonInit. You can see this in Table A‑2: ExplorerInit took longer when Slowsvc was removed from the load order group, because some beneficial prefetching activity no longer occurred.

Regardless of the position of the wait, a 6-second initialization time for a service is usually unacceptable.

## Sleep and Hibernate Transitions: Identifying Driver Delays in the SuspendDevices and ResumeDevices Phases

This example shows how to identify drivers that delay system suspend and resume because they respond slowly to system set-power IRPs or do not implement fast resume. Such drivers can significantly affect sleep and hibernate transition performance by delaying the system from entering suspend and causing serialization delays during resume.

#### Task 1: Observe driver suspend/resume times in the Summary.xml file

Follow the instructions in “Sleep and Hibernate Transitions: Capturing Traces” earlier in this paper to capture a sleep/hibernate trace and generate the Summary.xml file.

In the Summary.xml file, look under the **suspendDevices** and **resumeDevices** nodes. In the example in Figure A-9, the piwddev driver takes approximately 20 seconds in the SuspendDevices phase.

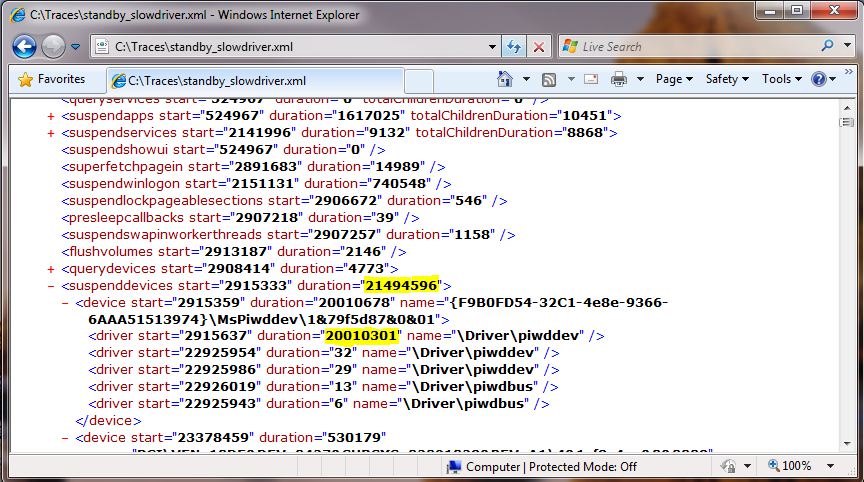


Figure A-9. Suspend trace in Summary.xml

#### Task 2: Scroll to the Generic Events graph and open the summary table

Open the trace file by using Xperfview and scroll down to the Generic Events graph. This graph displays all the kernel and user provider events that are captured in the trace. Select your region of interest, right-click, and choose **Summary Table**.

In Figure A-9, the piwddev driver starts at about 2.91 seconds and has a duration of approximately 20 seconds. Highlight this region in the Generic Events graph as shown in Figure A‑10.

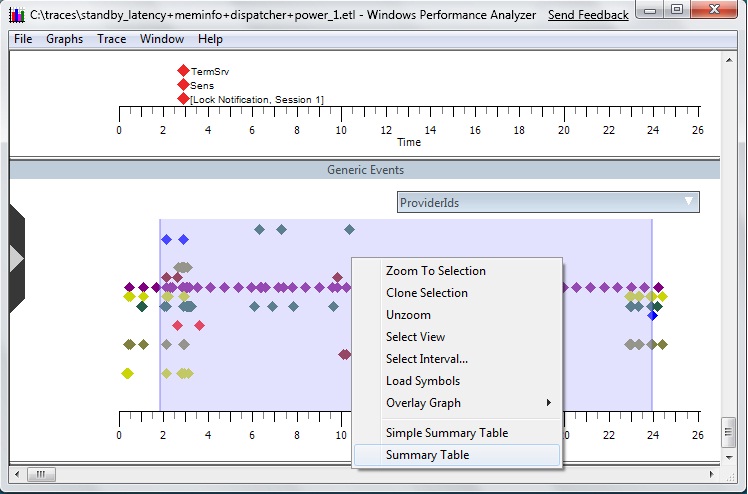


Figure A-10. Generic Events graph in Xperfview

Arrange the summary table as shown in Figure A-11, with Provider Name and Task Name columns as Grouping columns on the left, and Opcode Name, Field1 (Irp), Field6 (InstanceName), and Time(s) columns as Aggregate columns on the right.

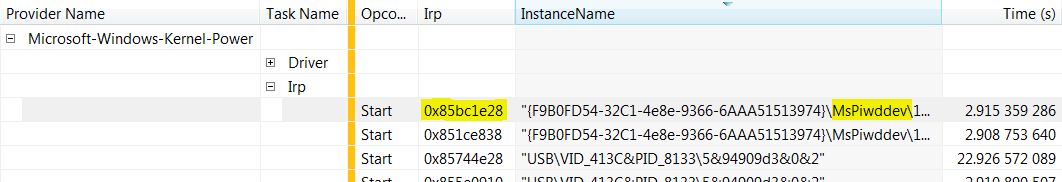


Figure A-11. Summary table for Microsoft-Windows-Kernel-Power provider

Under the Microsoft-Windows-Kernel-Power provider, expand the Irp task name and identify the driver that you are looking for. In Figure A‑11, the system has two instances of the Mspiwddev device and the first is highlighted.

For each instance of the device, record the value in the Irp column. Drag the Irp column to the left of the gold bar and scroll to the Irp value that you recorded. Click the box to expand this value, as shown in Figure A-12. You can now inspect the start and stop times to determine whether this device caused the delay. The IRP in Figure A-12 has a start time of 2.91 and a stop time of 22.926, so this IRP is likely the IRP that delayed system suspend. Note the thread ID (44) for your next task.

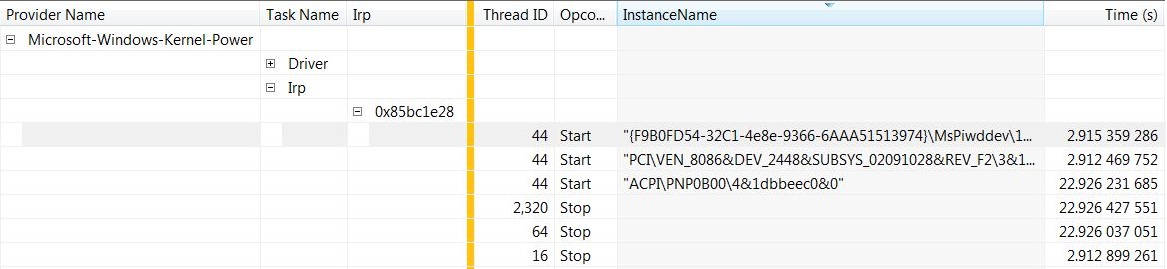


Figure A-12. Start and stop times for system power IRP

#### Task 3: Scroll to the CPU Scheduling graph and open the summary table with Ready Thread

If the CPU Scheduling graph is not visible, select it from the **Frame** list menu on the left. Ensure that symbols are loaded by clicking **Trace** and selecting **Load Symbols**. To set the symbol paths correctly, see “Symbol Support” on MSDN.

Highlight the region of interest in this graph, right-click, and select **Summary Table with Ready Thread** as shown in Figure A-13.

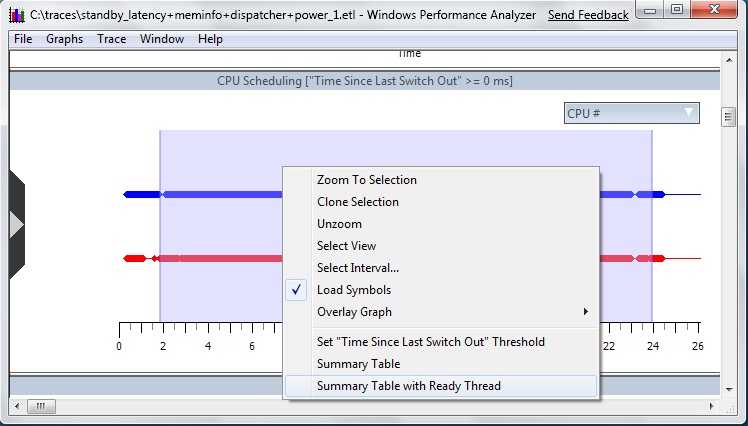


Figure A-13. CPU Scheduling graph

Arrange the summary table as shown in Figure A-14 with NewThreadID, NewProcess, SwitchInTime, NewThreadStack, ReadyThreadStack, ReadyingProcess, and ReadyingThreadID columns on the left of the gold bar and the Max:Waits column on the right.

In this example, we scroll to thread 44 as noted in the previous task, sort results in descending order of the Max:Waits(s) column, and expand the stacks for the first SwitchInTime.

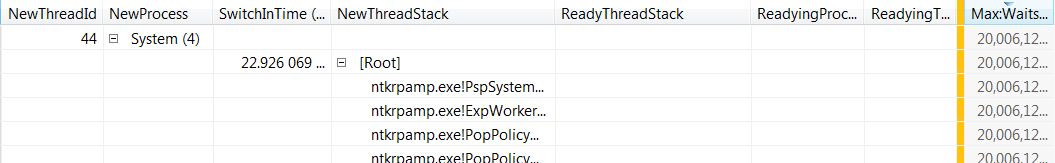


Figure A-14. Summary Table with Ready Thread

Continue to expand the stack until you find the readying stack in the ReadyThreadStack column that finally cleared this long wait.

As Figure A-15 shows, the readying stack belongs to thread 64 and also contains several calls to the piwddev.sys driver.

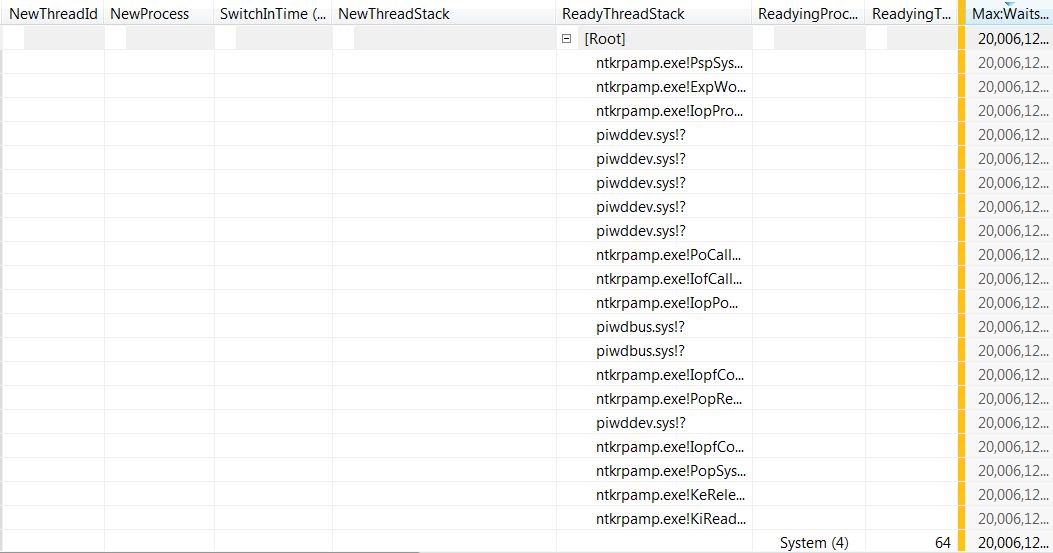


Figure A-15. Readying stack and thread

After you identify the readying thread, scroll to that thread ID in the NewThreadID column and look for the SwitchInTime that is closest to the original one but comes before it. In this example, the SwitchInTime of interest (Figure A-14) is approximately 22.926 seconds, so we scroll to thread 64 in the NewThreadID column and look for the closest previous value, which is approximately 22.925.

Expand the stacks under this column, identify the readying thread again, and repeat this process for each subsequent readying thread until you reach the cause of the long driver wait. The cause is the final ReadyThreadStack, which shows what the driver is waiting for. This appears as [Root] in Figure A‑16.

In this example, we followed this thread chain:

Thread 44 -> Thread 64 -> Thread 16 -> Thread 20

We finally identified the cause of this wait as a timer expiration. In other words, the driver had a 20-second timer that caused this delay, as shown in Figure A‑16.

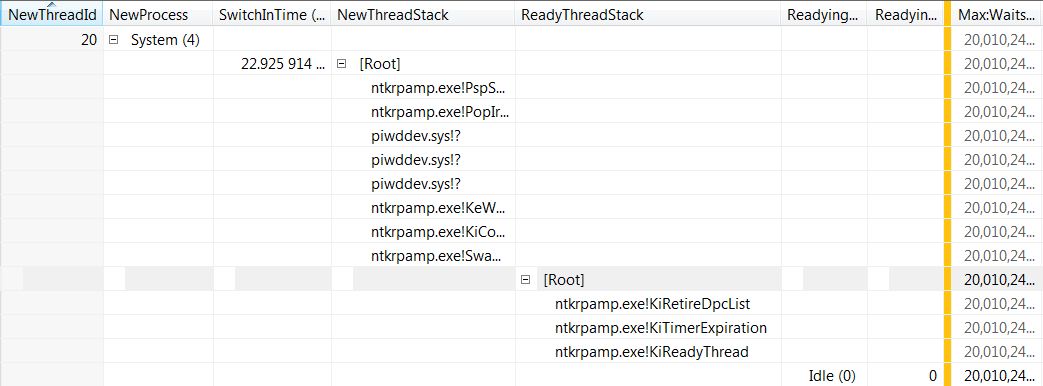


Figure A-16. Timer expiration causing delay

You can also follow the same steps to identify driver delays during system resume.

## Shutdown Phase: Measuring Application Delays in the UserSession Subphase

This example shows how to identify applications that delay system shutdown either by not responding at all or by responding slowly to shutdown notifications. Because Windows shuts down applications serially, every little delay in a process’s shutdown path contributes to the total shutdown duration.

#### Task 1: Observe application shutdown times in the Summary.xml file

Follow the instructions in “Shutdown Transitions: Capturing Traces” earlier in this paper to capture a sleep/hibernate trace and generate the Summary.xml file.

Look at the process shutdown times in the **sessionShutdown** node for sessionID = ‘1’ as Figure A-17 shows.



Figure A-17. sessionShutdown node in summary.xml

In this example, a single application that is called slow\_shutdown.exe takes approximately 5 seconds to shut down, which is almost 30 percent of the entire system shutdown time.

#### Task 2: Identify the slow application in Xperfview

After you identify the applications that delay shutdown, open the trace file in Xperfview.

Before you continue, ensure that symbols are loaded by clicking **Trace** and selecting **Load Symbols**. To set the symbol paths correctly, see “Symbol Support” on MSDN.

If the CPU Scheduling graph is not visible, select it from the **Frame** list on the left. Scroll to the CPU Scheduling graph, select the entire region of the graph, right-click this selection, and choose **Summary Table with Ready Thread** as shown in Figure A‑18.

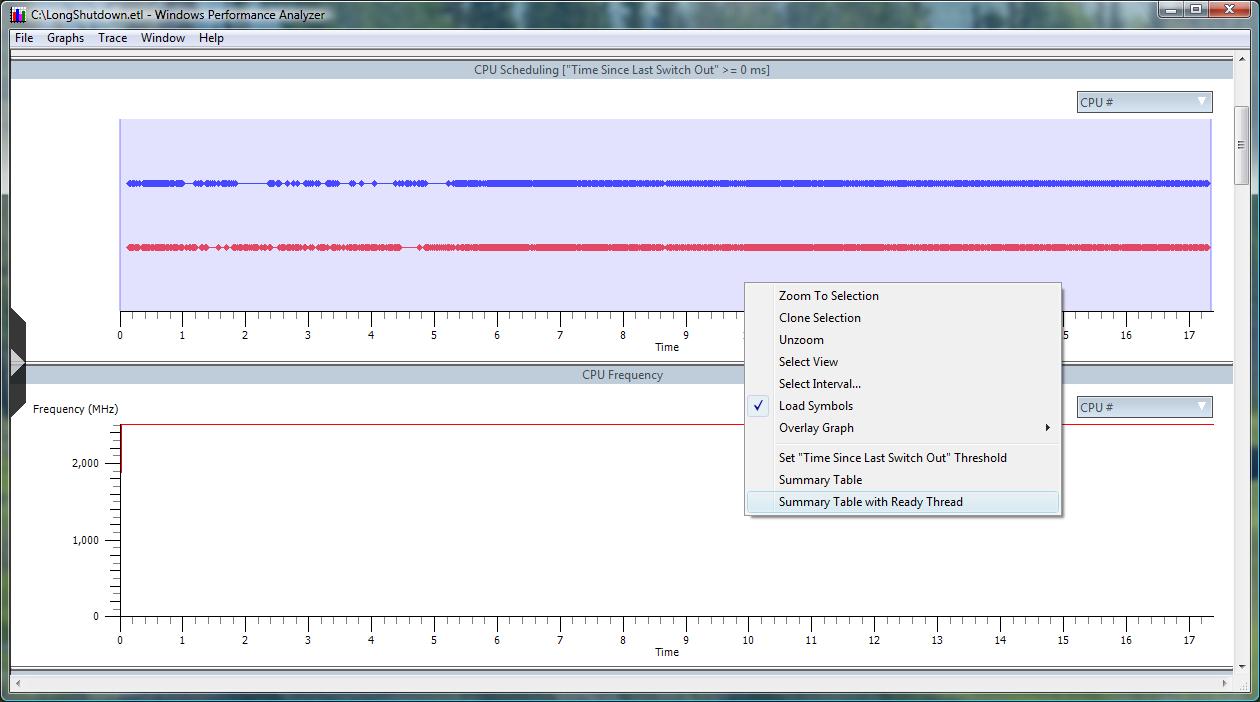


Figure A-18. CPU Scheduling graph

In the summary table, which is shown in Figure A-19, columns to the left of the gold bar are Grouping Columns and those to the right are Aggregated Columns.

For this example, we have the follow grouping columns in place: NewProcess, NewThreadStack, ReadyThreadStack, and ReadyingProcess.

After we sort the results by the NewProcess column, scroll to the unresponsive application (Slow\_shutdown.exe), and analyze the stacks, we see that this process does not respond to the shutdown notifications. Because Slow\_shutdown.exe is a console application, a 5-second time-out elapses before the client/server runtime subsystem (Csrss.exe) calls Winsrv.dll!KillProcess to close it.

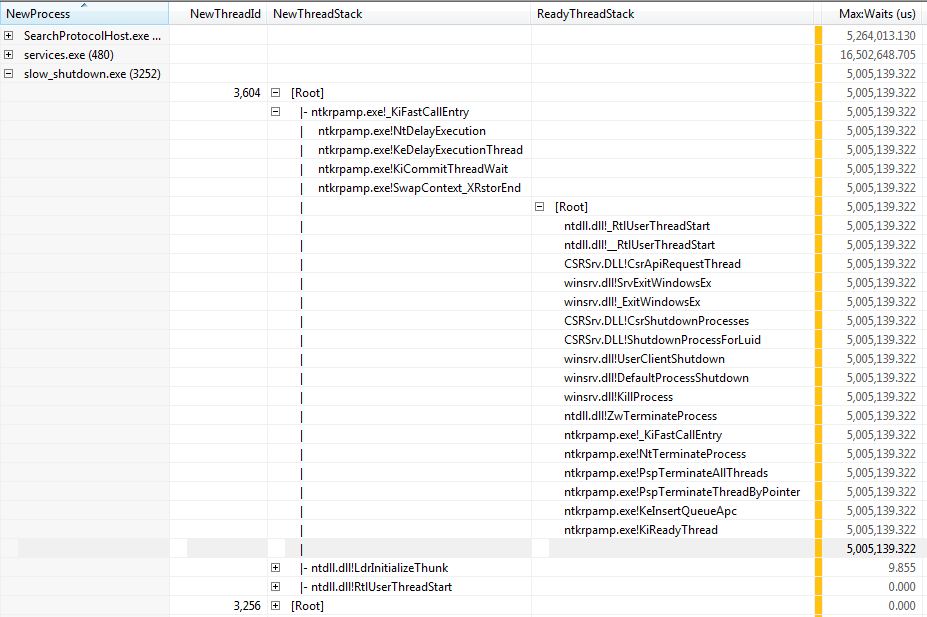


Figure A-19. Termination of Slow\_shutdown.exe process

Identification of the thread in this example is fairly straightforward because the application has only two threads and all delays occurred in the first thread (ThreadId=3,604). However, real-world applications usually have several threads and waits, so that it is not as easy to identify the cause. By using the Wait Classification feature in WPT v4.6, you can quickly filter unrelated waits and focus only on waits that occur in threads where the waits can affect end-user perceived performance. In addition, you can further narrow your focus within those threads, to waits that occur only in functions that should not be blocked for long. WPT 4.6 is available in the Windows 7 Software Development Kit (SDK), which is listed in “Resources.”

## All Transitions: Identifying Applications That Consume Significant CPU Time

This example shows how to identify applications that consume significant CPU resources either during an on/off transition or when the system is idle. Applications should minimize their use of resources such as disk and CPU at all times to improve system responsiveness and user perceptions of system performance.

#### Task 1: Open the trace file in Xperfview and scroll to the CPU Sampling graph

Capture a trace by using Xperf, and open it in Xperfview either by double-clicking the trace file or by typing the following at a command prompt:

xperf <tracefile.etl>

Before you continue, ensure that symbols are loaded by clicking **Trace** and selecting **Load Symbols**. To set the symbol paths correctly, see “Symbol Support” on MSDN.

Scroll to the CPU Sampling graph as shown in Figure A-20. Select the entire duration or a specific region of interest in the trace, right-click, and then select **Summary Table**from the menu. In Figure A-20, a 6-second period is selected at the upper left.

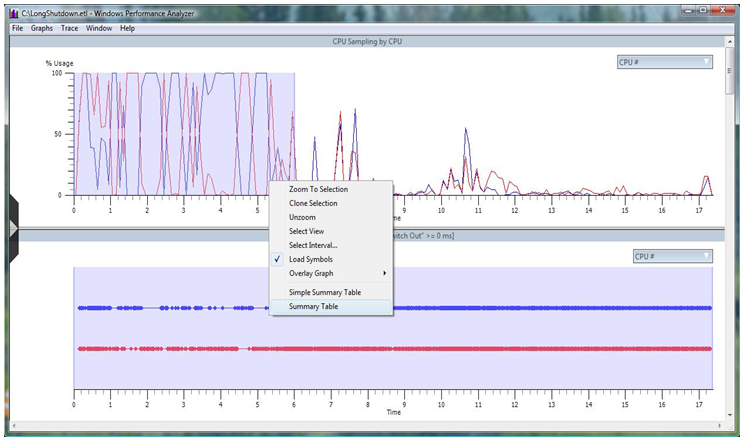


Figure A-20. CPU Sampling graph

#### Task 2: Identify the applications that consume CPU in the summary table

In the summary table, columns to the left of the golden bar are Grouping Columns and those to the right are Aggregate Columns. You can drag and drop columns on either side, and you can add more columns from the Column Chooser menu on the left or the Columns menu on the top.

In Figure A-21, you can see that apart from System Idle, the Slow\_shutdown.exe process is responsible for most of the CPU usage during the 6 seconds that were highlighted in Figure A‑20. The Weight and %Weight columns in the gray row show that this process used about 44 percent of the CPU for about 5.22 seconds. If you expand the stacks, you can see that the process was stuck in a loop for this entire duration.

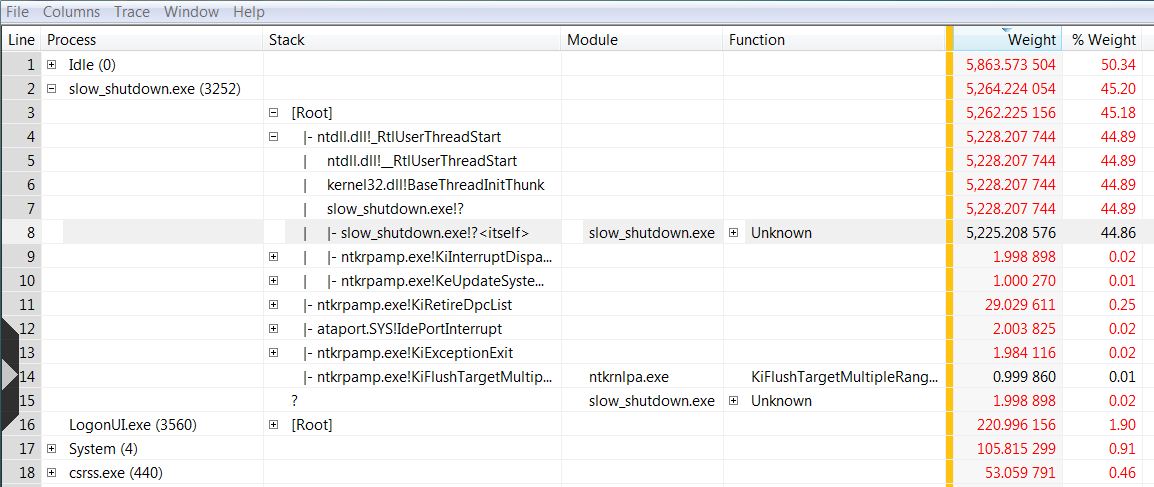


Figure A-21. Summary table that shows CPU use

If you want to view only the CPU use of a specific application, you can use the CPU Sampling by Process graph, as shown in Figure A-22. By using the filter menu on the right, you can select a process, observe its CPU usage in the graph, and view the corresponding summary table for more details.

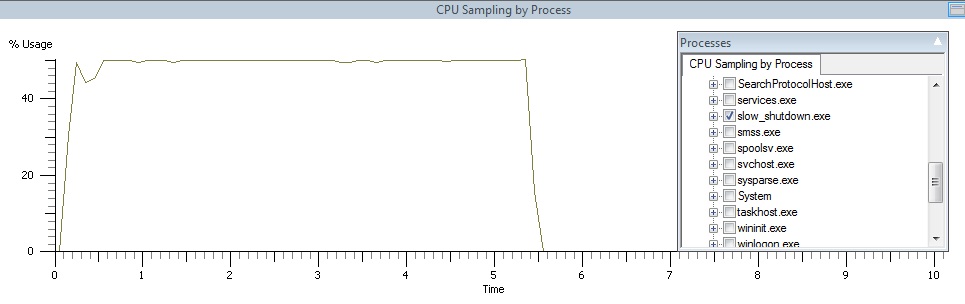


Figure A-22. CPU use for a specific process

# Appendix B. Raw Event Dumps

The Xperf tool can generate a raw per-event dump of a trace, which provides detailed event information. This section briefly describes how to create and inspect a raw dump file.

First, generate a trace and name it “trace.etl”.

Then create the dump by typing the following at the command prompt:

xperf –i trace.etl –o trace.csv –a dumper

As you can see in Figure B-1, the resulting dump file is detailed and can be difficult to understand unless you are familiar with the system and events.

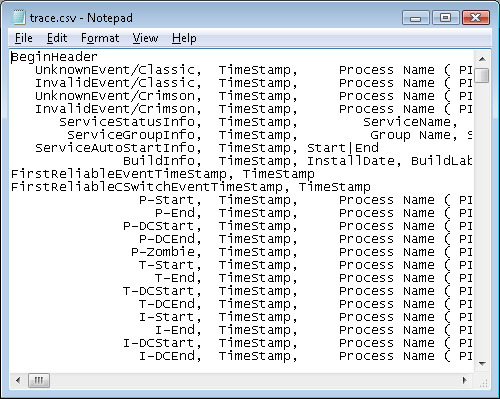


Figure B-1. A sample dump file that Xperf created

The general format of the dump is as follows:

* The first column represents the event type.
* The second column represents the timestamp of the event in microseconds since the start of the trace.
* Any remaining columns contain event-specific information.

Raw dump files are often tens or hundreds of MBs and require filtering for effective analysis. The easiest way to filter the file is to use a utility such as Findstr.exe, which is included with Windows Vista and Windows 7.

The Findstr.exe arguments follow regular expression syntax. For command-line help, type the following:

findstr.exe /?

For example, to obtain a list of all events that refer to Winusb.dll in the Trace.csv dump file, type the following:

findstr winusb.dll trace.csv

# Appendix C. .CSV Summaries of Plug and Play and Service Activity

If you generate a .CSV summary of the Plug and Play activity in the boot trace, you can view trace data in an Excel spreadsheet.

To generate the .csv file, use the **-a pnp** action with Xperf. For example, the following command takes as input the Trace.etl trace file and generates the Pnpsummary.csv output file:

xperf -i trace.etl -o pnpsummary.csv -a pnp

If you open Pnpsummary.csv, you should see four event types in the **EventType** column. Column A in Figure C-1 shows three of these event types.

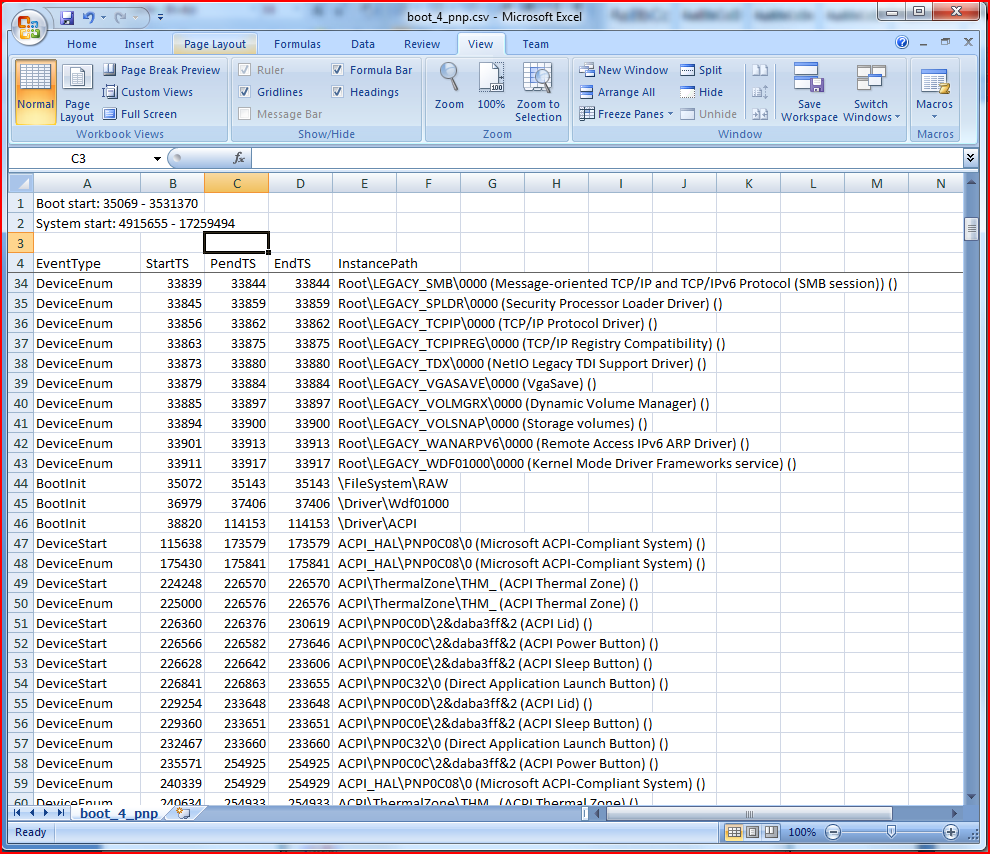


Figure C-1. CSV summary of Plug and Play activity

The following four event types can appear in the EventType column:

* **BootInit** represents the required time to call the **DriverEntry** functions of BOOT\_START drivers that were previously loaded during the OSLoader phase. These drivers do not have associated DriverLoad events.
* **DriverLoad** represents the loading process and call to **DriverEntry** for non‑BOOT\_START drivers.
* **DeviceStart** represents the required time to process the IRP\_MN\_START\_DEVICE IRP.
* **DeviceEnum** represents the required time to process the IRP\_MN\_QUERY\_DEVICE\_RELATIONS IRP.

Each Plug and Play operation has three timestamps, all of which are measured from the beginning of the trace:

* **StartTS** is the time at which the operation started.
* **PendTS** is the time at which the target pended the operation. For operations that are not pended, PendTS is set to EndTS.
* **EndTS** is the time at which the operation completed.

Each operation has an associated **InstancePath** attribute that corresponds to a symbolic name for a device or driver.

The Summary Plug and Play data, which is located toward the end of the .csv file, contains separate sections for per-device and per-driver results. It calculates the durations from the raw data.

Another common area of investigation for boot performance issues is service initialization and startup times. To generate a .csv summary of service activity in the boot trace, use the **-a services** action with Xperf. For example, the following command takes as input the trace.etl trace file and generates a .csv output file:

xperf -i trace.etl -o servicesummary.csv -a services

Figure C-2 shows the .csv output in Servicesummary.csv.

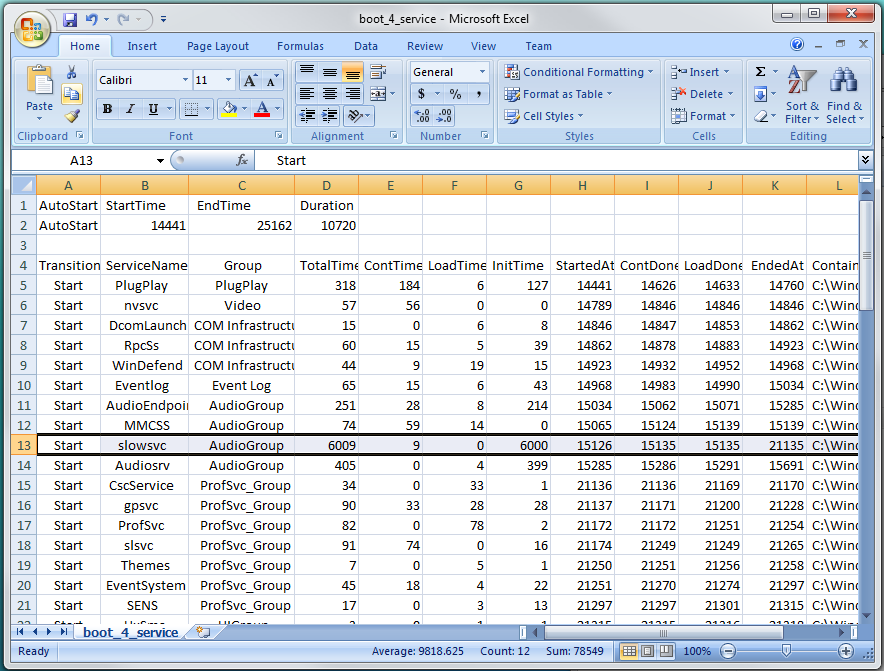


Figure C-2. CSV summary of service activity

AutoStart service timing metrics such as start, stop, and elapsed time appear at the top of the file. These metrics show the time in milliseconds that was required to complete the transition for all autostart services.

The columns in the Servicesummary.csv file are as follows:

* **Transition** contains the type of service transition, such as Start or Stop.
* **ServiceName** is the name of the service. The name is the actual service name, not the long display name that appears in Services.msc.
* **Group** displays the load order group with which the service is associated. This column is blank if the service is not associated with any group.
* The **TotalTime**, **ContTime**, **LoadTime**, and **InitTime** columns display the elapsed time to complete each service operation.
* The **StartedAt**, **ContDoneAt**, **LoadDoneAt**, and **EndedAt** columns display the timestamp in milliseconds of each service operation compared with the start of the trace (t=0).
* **Container** shows a full path to the executable image that started the service.

In this particular trace, Slowsvc clearly stands out at 6 seconds as a performance problem. You can confirm that the service belongs to AudioGroup and obtain precise timestamps of the start and end times from this output.

These summaries can help you identify devices, drivers, and services that have a large negative effect on boot times. The summaries can also help you determine how much time is required to load and initialize a particular device and driver combination or set of services.