Driver Verifier in Windows 7

November 5, 2008

Abstract

Driver Verifier monitors kernel-mode drivers to detect incorrect function calls or actions that might corrupt the system. In Windows® 7, Driver Verifier has several features that did not exist in earlier Windows versions, that detect new classes of driver defects, and that provide information for debugging these driver defects. This paper provides a preview of the Windows Driver Kit (WDK) documentation for these enhancements. Exposing these additional classes of common driver bugs assists driver developers in producing higher quality device drivers.

This information applies for the Windows 7 operating system.

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

For the latest information, see:
 <http://www.microsoft.com/whdc/devtools/tools/Win7SDV.mspx>

Disclaimer: This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2008 Microsoft Corporation. All rights reserved.

Microsoft, MSDN, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Change |  |  |  |
| November 5, 2008 | First publication |

Contents

[Introduction 4](#_Toc212800343)

[Incorrect References to User Handles from Kernel Drivers 4](#_Toc212800344)

[I/O Verification Improvements 6](#_Toc212800345)

[Special Pool, Pool Tracking, and Low Resources Simulation Improvements 7](#_Toc212800346)

[Incorrect Usage of Synchronization Mechanisms 10](#_Toc212800347)

[UserMode Wait for Synchronization Objects Allocated on the Kernel Stack 10](#_Toc212800348)

[Synchronization Object Allocation from NonPagedPoolSession Memory 11](#_Toc212800349)

[**KeEnterCriticalRegion** or **KeLeaveCriticalRegion** Calls at
 DISPATCH\_LEVEL or Above 11](#_Toc212800350)

[Incorrect Usage of InStackQueuedSpinlock APIs 12](#_Toc212800351)

[Incorrect Object References 13](#_Toc212800352)

[Object Reference Counter Changes from 0 to 1 13](#_Toc212800353)

[Incorrect Kernel Handle References 13](#_Toc212800354)

[Pool Quota Charges from DPC Routine 14](#_Toc212800355)

[System Shutdown Blocks or Delays 15](#_Toc212800356)

[Improved Force Pending I/O Requests 15](#_Toc212800357)

[Additional Debugging Information 16](#_Toc212800358)

[Resources 17](#_Toc212800359)

# Introduction

Driver Verifier monitors kernel-mode drivers to detect incorrect function calls or actions that might corrupt the system. It can subject the drivers to many different stresses and tests to find incorrect behavior.

We strongly encourage hardware manufacturers to test their drivers by using Driver Verifier to ensure that drivers are not making incorrect function calls or causing system corruption. Driver Verifier is included in Windows® 2000 and all later versions of Windows. Driver Verifier works on both free and checked builds of Windows.

For Windows 7, Driver Verifier has several new tests and features that allow it to expose more classes of typical driver bugs. Also, Driver Verifier provides more information that is useful for debugging these driver defects. This paper provides a preview of Windows Driver Kit (WDK) documentation for the Driver Verifier enhancements in Windows 7.

Debugging Tools for Windows are typically used to investigate issues that Driver Verifier exposes, as mentioned in this preview. For availability of Windows 7 and Debugging Tools for Windows, see “Resources” at the end of this paper.

# Incorrect References to User Handles from Kernel Drivers

Each Windows process has a handle table. One can view the handle table as an array of handle entries. Each valid handle value refers to a valid entry in this array.

We define a *kernel handle* as a handle that is valid for the System process’s handle table. The System process is the process in which threads that were created by **PsCreateSystemThread** and system worker threads run. The System process’s identifier is 4. The user can display information about it in the kernel debugger by using the **!process 4** command.

We define a *user handle* as a handle that is valid in any process except the System process.

When a system thread creates a handle—for example by calling **ZwCreateFile**—the resulting handle is a kernel handle. A kernel driver can also create kernel handles while it is running in the context of any other process by calling one of the Windows kernel APIs—such as **ZwOpenEvent**—and specifying OBJ\_KERNEL\_HANDLE as one of the OBJECT\_ATTRIBUTES for that API call.

Generally, user handles are created by an application. For example, calling **CreateEvent** can create an event handle, and calling **CreateProcess** can create two handles: one for a process and another for a thread object. However, when a kernel driver creates a handle while it is running in the context of an application and without specifying the OBJ\_KERNEL\_HANDLE attribute, the result is also a user handle. The driver must be very careful when it uses such handles because the application can:

* Use the handle to reference the corresponding object.
* Close the handle while the driver still needs it.
* Replace the handle with a new handle that points to a different object.

Kernel drivers can reference handles with an *AccessMode* that is either **UserMode** or **KernelMode**. Examples of **KernelMode** references include the following:

* Calling **ObReferenceObjectByHandle** with the *AccessMode*parameter set to **KernelMode**.
* Calling a **Zw*Xxx*** function, passing the handle as a parameter, for example, calling **ZwSetEvent**.

**KernelMode** references must be used carefully because they have the following properties:

* The *DesiredAccess* parameter is ignored. This property means that the handle reference succeeds regardless of the security access that was granted to the current process when the handle was created. For example, a **KernelMode** reference for a write to a read-only file handle succeeds.
* **KernelMode** references are allowed to reference kernel handle values, regardless of the current process context.

In Windows 7, Driver Verifier detects attempts to reference user handle values from function calls with *AccessMode* set to **KernelMode**. This test exposes typical driver defects, such as:

* Creating a handle but forgetting to specify the OBJ\_KERNEL\_HANDLE attribute. This action can be dangerous because the application can use that handle, close it, or replace it with another handle that has the same value but points to a different object.
* Receiving a handle from an application and referencing it blindly as **KernelMode**. This action is incorrect because the reference succeeds regardless of the access that was granted to the application when the handle was created. Also, the application could be sending a kernel handle value to the driver, and the **KernelMode** reference on behalf of the application succeeds. Applications should never be allowed to use kernel handles.

However, Driver Verifier ignores **KernelMode** references to user handles if these references occur in the context of a process that runs under the local system account. Because these processes hold powerful privileges, they are less likely to pose a security risk.

Driver Verifier Security Checks must be enabled to expose these driver defects. You can activate the Security Checks option for one or more drivers by using Driver Verifier manager or the Verifier.exe command line. At the command line, the Security Checks option is represented by bit 8 (0x100).

To activate the Security Checks option at the command line

* Use a flag value of 0x100 or add 0x100 to the flag value, as shown in the following example:

verifier /flags 0x100 /driver MyDriver.sys

The option is active after the next boot.

You can also activate and deactivate Security Checks without rebooting the computer by adding the */volatile* parameter to the command, as shown in the following example:

verifier /volatile /flags 0x100 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

The Security Checks option is also included in the /standard settings, as shown in the following example:

verifier /standard /driver MyDriver.sys

To activate Security Checks by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select Security Checks.

# I/O Verification Improvements

The Enhanced I/O Verification option of Driver Verifier is no longer available in Windows 7. Most of the checks that were previously performed when Enhanced I/O Verification was enabled are now performed when the “regular” I/O Verification is enabled.

In Windows versions earlier than Windows 7, Driver Verifier standard settings did not include the Enhanced I/O Verification option. However, the “regular” I/O Verification is included in standard settings. This means that in Windows 7 users who enable standard settings during their testing but do not enable Enhanced I/O Verification can expose many additional classes of driver bugs. For more information about these additional classes of bugs, see the “Enhanced I/O Verification” topic on MSDN®.

In Windows 7, Driver Verifier I/O Verification can detect another common driver bug—reinitializing remove locks. Remove locks data structures should be allocated inside device extensions. This ensures that the I/O manager frees the memory that holds the IO\_REMOVE\_LOCK structure only when the device object is deleted. If the driver performs the following three steps, it is possible that after step 2 an application or driver still holds a reference to Device1:

1. Allocates the IO\_REMOVE\_LOCK structure that corresponds to Device1 but outside Device1’s extension.

2. Calls **IoReleaseRemoveLockAndWait** when Device1 is being removed.

3. Calls **IoInitializeRemoveLock** for the same lock to reuse it as a remove lock for Device2.

It is possible that after step 2 an application or driver still holds a reference to Device1. The application or driver can still send requests to Device1, even though this device was removed. Therefore, it is not safe to reuse the same memory as a new remove lock until I/O manager deletes Device1. Reinitializing the same lock while another thread is trying to acquire it can result in the corruption of the lock, with unpredictable results for the driver and the entire system.

You can activate the I/O Verification option for one or more drivers by using Driver Verifier manager or the Verifier.exe command line. At the command line the I/O Verification option is represented by Bit 4 (0x10).

To activate I/O Verification at the command line

* Use a flag value of 0x10 or add 0x10 to the flag value, as shown in the following example:

verifier /flags 0x10 /driver MyDriver.sys

The option is active after the next boot.

You can also activate and deactivate I/O Verification without rebooting the computer by adding the */volatile* parameter to the command, as shown in the following example:

verifier /volatile /flags 0x10 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

The I/O Verification option is also included in the /standard settings, as shown in the following example:

verifier /standard /driver MyDriver.sys

To activate I/O Verification by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select I/O verification.

To use the I/O Verification option in the standard settings of Driver Verifier manager, click Create Standard Settings.

# Special Pool, Pool Tracking, and Low Resources Simulation Improvements

Special Pool is a Driver Verifier option that can expose typical memory corruptions, such as using memory after free or allocating *n* bytes and then using more than *n* bytes.

Pool Tracking monitors the memory allocations that are made by the driver. When the driver is unloaded, Driver Verifier ensures that all allocations that the driver made were freed. Nonfreed memory allocations (also known as memory leaks) are a common cause of lowered operating system performance. Memory leaks can fragment the system pools and eventually cause system crashes.

When the Low Resources Simulation option is active, Driver Verifier fails random instances of the driver's memory allocations, as might occur if the driver is running on a computer that has insufficient memory. This action tests the driver's ability to respond correctly to low-memory and other low-resource conditions.

Special Pool, Pool Tracking, and Low Resources Simulation options are effective mechanisms for exposing driver defects that are related to memory blocks that are allocated by using common kernel APIs such as **ExAllocatePoolWithTag**. For additional information about these Driver Verifier options, see “Driver Verifier Options” on MSDN.

In Windows 7, Driver Verifier is improved to provide Special Pool, Pool Tracking, and Low Resources Simulation functionality for memory that was allocated by using the following kernel APIs:

* **IoAllocateMdl**.
* **IoAllocateIrp** and the other APIs that can allocate I/O request packet (IRP) data structures.
* **RtlAnsiStringToUnicodeString** and other **Rtl** String APIs.
* **IoSetCompletionRoutineEx**.

You can activate the Special Pool, Pool Tracking, and Low Resources Simulation options for one or more drivers by using Driver Verifier manager or the Verifier.exe command line. At the command line:

* The Special Pool option is represented by Bit 0 (0x1).
* The Pool Tracking option is represented by Bit 3 (0x8).
* The Low Resources Simulation option is represented by Bit 2 (0x4).

To activate Special Pool at the command line

* Use a flag value of 0x1 or add 0x1 to the flag value, as shown in the following example:

verifier /flags 0x1 /driver MyDriver.sys

The option is active after the next boot.

You can also activate and deactivate Special Pool without rebooting the computer by adding the */volatile* parameter to the command, as shown in the following example:

verifier /volatile /flags 0x1 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

The Special Pool option is also included in the /standard settings, as shown in the following example:

verifier /standard /driver MyDriver.sys

To activate Special Pool by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select Special Pool.

To use the Special Pool option in the standard settings, in Driver Verifier manager, click Create Standard Settings.

To activate Pool Tracking at the command line

* Use a flag value of 0x8 or add 0x8 to the flag value, as shown in the following example:

verifier /flags 0x8 /driver MyDriver.sys

The option is active after the next boot.

You can also activate and deactivate Pool Tracking without rebooting by adding the */volatile* parameter to the command, as shown in the following example:

verifier /volatile /flags 0x8 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

The Pool Tracking option is also included in the /standard settings, as shown in the following example:

verifier /standard /driver MyDriver.sys

To activate Pool Tracking by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select Pool Tracking.

To use the Pool Tracking option in the standard settings for Driver Verifier manager, click Create Standard Settings.

To activate Low Resources Simulation at the command line

* Use a flag value of 0x4 or add 0x4 to the flag value as shown in the following example:

verifier /flags 0x4 /driver MyDriver.sys

The option is active after the next boot.

You can also activate and deactivate Pool Tracking without rebooting by adding the */volatile* parameter to the command, as shown in the following example:

verifier /volatile /flags 0x4 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

The Pool Tracking option is not included in the /standard settings.

To activate Low Resources Simulation by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select Low Resources Simulation.

# Incorrect Usage of Synchronization Mechanisms

In Windows 7, Driver Verifier can detect several additional incorrect ways to use the multithreading synchronization mechanisms that the operating system provides.

## UserMode Wait for Synchronization Objects Allocated on the Kernel Stack

Allocating synchronization objects, such as KEVENT structures, as local variables on the kernel stack is a common practice. While a process is loaded in memory, the kernel stacks of its threads are never trimmed from the working set or paged out to the disk. Allocating synchronization objects in such nonpageable memory is correct.

However, when drivers call APIs such as **KeWaitForSingleObject** or **KeWaitForMultipleObjects** to wait for an object that is allocated on the stack, they must specify the **KernelMode** value for the API’s *WaitMode* parameter. When all threads of a process are waiting in **UserMode** mode, that process becomes eligible to be swapped out to the disk. Therefore, if a driver specified **UserMode** as the *WaitMode* parameter, the operating system can swap out the current process as long as every other thread in the same process is waiting as **UserMode**, too. Swapping an entire process out to the disk includes paging out its kernel stacks. Waiting on a synchronization object that the operating system has swapped out is incorrect. At some point a thread must come along and signal the synchronization object. Signaling a synchronization object involves the Windows kernel manipulating the object at IRQL = DISPATCH\_LEVEL or above. Touching paged out or swapped out memory at DISPATCH\_LEVEL or above results in a system crash.

Enabling Driver Verifier Miscellaneous Checks in Windows 7 results in checking that synchronization objects that a verified driver uses for waiting in **UserMode** are not allocated on the current thread’s kernel stack. When Driver Verifier detects such an incorrect wait, it issues a bug check code 0xC4, subcode 0x123.

You can activate the Miscellaneous Checks option for one or more drivers by using Driver Verifier manager or the Verifier.exe command line. At the command line the Miscellaneous Checks option is represented by bit 11 (0x800).

To activate Miscellaneous Checks at the command line

* Use a flag value of 0x800, or add 0x800 to the flag value, as shown in the following example:

verifier /flags 0x800 /driver MyDriver.sys

The option is active after the next boot.

You can also activate and deactivate Miscellaneous Checks without rebooting by adding the /volatile parameter to the command, as shown in the following example:

verifier /volatile /flags 0x800 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

The Miscellaneous Checks option is also included in the /standard settings, as shown in the following example:

verifier /standard /driver MyDriver.sys

To activate Miscellaneous Checks by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select Miscellaneous Checks.

## Synchronization Object Allocation from NonPagedPoolSession Memory

Synchronization objects must be nonpageable. They must also live in the global, system-wide virtual address space.

A graphics driver can allocate session memory by calling APIs such as **EngAllocMem**. Unlike the global address space, the session address space is virtualized for each Terminal Server session. This means that the same virtual address that is used in the context of two different sessions refers to two different objects. The Windows kernel must be able to access synchronization objects from any Terminal Server session. Trying to reference a session memory address from a different session has unpredictable results, such as system crashes or silent corruption of another session’s data.

In Windows 7, when a verified driver initializes a synchronization object by calling APIs such as **KeInitializeEvent** or **KeInitializeMutex**, Driver Verifier checks whether the address of the object falls inside the session virtual address space. If Driver Verifier detects this kind of incorrect address, it issues a bug check code 0xC4, subcode 0xDF.

These checks for synchronization objects that are allocated from session memory are enabled regardless of the Driver Verifier options that are currently enabled. Enabling any Driver Verifier option enables these checks.

## **KeEnterCriticalRegion** or **KeLeaveCriticalRegion** Calls at DISPATCH\_LEVEL or Above

**KeEnterCriticalRegion** and **KeLeaveCriticalRegion** are APIs that can be used to synchronize the execution of a critical sequence of driver code with the delivery of ordinary kernel asynchronous procedure calls (APCs). The MSDN documentation for these APIs (**KeEnterCriticalRegion** and **KeLeaveCriticalRegion**) specifies that they cannot be called at IRQL = DISPATCH\_LEVEL or above. Calling **KeEnterCriticalRegion** or **KeLeaveCriticalRegion** at DISPATCH\_LEVEL or above can result in a system hang or memory corruption.

In Windows 7, Driver Verifier detects calls to these APIs at DISPATCH\_LEVEL or above if the Force IRQL Checking option is enabled.

You can activate the Force IRQL Checking option for one or more drivers by using Driver Verifier manager or the Verifier.exe command line. At the command line the Force IRQL Checking option is represented by Bit 1 (0x2).

To activate Force IRQL Checking at the command line

* Use a flag value of 0x2 or add 0x2 to the flag value, as shown in the following example:

verifier /flags 0x2 /driver MyDriver.sys

The option is active after the next boot.

You can also activate and deactivate Force IRQL Checking without rebooting by adding the */volatile* parameter to the command, as shown in the following example:

verifier /volatile /flags 0x2 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

The Force IRQL Checking option is also included in the /standard settings, as shown in the following example:

verifier /standard /driver MyDriver.sys

To activate Force IRQL Checking by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select Force IRQL Checking.

To use this option in the standard settings for Driver Verifier manager, click Create Standard Settings.

## Incorrect Usage of InStackQueuedSpinlock APIs

In Windows 7, Driver Verifier provides **InStackQueuedSpinlock** API checks that resemble those provided to “regular” spinlocks in earlier Windows versions. These checks include the following:

* Verifying that an operation that should raise the interrupt request level (IRQL) value, such as **KeAcquireInStackQueuedSpinLock**, is not actually lowering the IRQL value.
* Verifying that an operation that should lower the IRQL value, such as **KeReleaseInStackQueuedSpinLock**, is not actually raising the IRQL value.
* Trimming the System process’s working set if the IRQL option is enabled, when the IRQL is being raised to DISPATCH\_LEVEL or above, in an attempt to expose possible references to pageable memory while the driver is running at elevated IRQL.
* Predicting possible deadlocks when the Deadlock Detection option is enabled.
* Trying to use the same KSPIN\_LOCK data structure both as a “regular” spinlock and as a stack queued spinlock when the Deadlock Detection option is enabled.
* Checking for obviously incorrect pointer values, such as a user-mode virtual address that is used as a spinlock address.
* Logging IRQL transitions in the Driver Verifier IRQL log. This log appears with the **!verifier 8** extension of the Windows Debuggers.

# Incorrect Object References

In Windows 7, Driver Verifier can detect incorrect additional classes of incorrect object references from verified drivers.

## Object Reference Counter Changes from 0 to 1

When the Windows kernel object manager creates an object, such as a File object or a Thread object, the new object’s reference counter is set to 1. The reference counter is incremented by calls to APIs such as **ObReferenceObjectByPointer** or **ObReferenceObjectByHandle**. The reference counter is decremented by every **ObDereferenceObject** call for the same object.

After the reference counter reaches the 0 value, the object becomes eligible to be freed. The object manager might free it immediately, or it might free it later. Calling **ObReferenceObjectByPointer** or **ObReferenceObject** and changing the reference counter from 0 to 1 means incrementing the reference counter of an already freed object. This is always incorrect because it can result in corrupting someone else’s memory allocation.

In Windows 7, Driver Verifier detects such incorrect API calls from verifier drivers. These checks are enabled regardless of the Driver Verifier options that are currently enabled. Enabling any Driver Verifier option enables these checks.

## Incorrect Kernel Handle References

Each Windows process has a handle table. You can view the handle table as an array of handle entries. Each valid handle value refers to a valid entry in this array.

We define a *kernel handle* as a handle that is valid for the System process’s handle table. We define a *user handle* as a handle that is valid for any process except the System process.

In Windows 7, Driver Verifier detects tries to reference kernel handle values that are incorrect. These driver defects are reported as a bug check code 0x93 if the Driver Verifier Miscellaneous Checks option is enabled. Usually this kind of incorrect handle reference means that the driver has closed that handle already but is trying to continue using it. This kind of defect can result in unpredictable problems for the system because the handle value that is being referenced might have been reused already by another unrelated driver.

If a kernel driver has recently closed a kernel handle and later references the closed handle, Driver Verifier forces the bug check as described previously. In this case the output of the **!htrace** debugger extension provides the stack trace for the code path that closed this handle. Use the address of the System process as a parameter for **!htrace**. To find the address of the System process, use the **!process 4 0** command.

You can activate the Miscellaneous Checks option for one or more drivers by using Driver Verifier manager or the Verifier.exe command line. At the command line, the Miscellaneous Checks option is represented by bit 11 (0x800).

To activate Miscellaneous Checks at the command line

* Use a flag value of 0x800 or add 0x800 to the flag value, as shown in the following example:

verifier /flags 0x800 /driver MyDriver.sys

The option is active after the next boot.

You can also activate and deactivate Miscellaneous Checks without rebooting by adding the /volatile parameter to the command, as shown in the following example:

verifier /volatile /flags 0x800 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

The Miscellaneous Checks option is also included in the /standard settings, as shown in the following example:

verifier /standard /driver MyDriver.sys

To activate Miscellaneous Checks by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select Miscellaneous Checks.

# Pool Quota Charges from DPC Routine

Kernel drivers can call **ExAllocatePoolWithQuotaTag** to allocate kernel pool memory and charge the number of bytes that are allocated to the pool quota of the current process. Drivers typically use quota for memory allocations that are directly related to a request that comes from an application.

Deferred procedure call (DPC) routines can run in the context of any process. Therefore, charging quota from a DPC routine charges a random process. Even worse, when the DPC routine runs in the context of the Idle process, this condition can result in memory corruption or system crashes.

In Windows 7, Driver Verifier detects **ExAllocatePoolWithQuotaTag** calls from DPC routines if the Pool Tracking option is enabled.

To activate Pool Tracking at the command line

* Use a flag value of 0x8 or add 0x8 to the flag value, as shown in the following example:

verifier /flags 0x8 /driver MyDriver.sys

The option is active after the next boot.

You can activate and deactivate Pool Tracking without rebooting by adding the /volatile parameter to the command, as shown in the following example:

verifier /volatile /flags 0x8 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

The Pool Tracking option is also included in the /standard settings, as shown in the following example:

verifier /standard /driver MyDriver.sys

To activate Pool Tracking by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select Pool Tracking.

To use the Pool Tracking option in the Driver Verifier standard settings, click Create Standard Settings.

# System Shutdown Blocks or Delays

In Windows 7, Driver Verifier issues a break into the kernel debugger if the system shutdown does not finish 20 minutes after it started. Driver Verifier assigns the start of the system shutdown as the time when the Windows kernel begins shutting down its various subsystems, such as the Registry, Plug And Play, or the I/O manager subsystems.

If a kernel debugger is not attached to the system, Driver Verifier issues a bug check code 0xC4, subcode 0x115, instead of this breakpoint.

Frequently a system shutdown that cannot finish in less than 20 minutes indicates that one of the drivers that is running on that system is malfunctioning. Running **!analyze -v** from the kernel debugger displays the stack trace of the system worker thread that is responsible for the shutdown. The Driver Verifier user should examine that stack trace and determine whether the shutdown thread is blocked by one of the drivers that are being tested.

Sometimes the system cannot shut down because it is subject to heavy stress testing—even though all drivers are functioning correctly. The user can choose to continue the execution after this Driver Verifier breakpoint and check whether the system eventually shuts down.

This Driver Verifier break occurs regardless of the options that are currently enabled. Enabling any Driver Verifier option enables this breakpoint.

# Improved Force Pending I/O Requests

The Driver Verifier Force Pending I/O Requests option randomly returns STATUS\_PENDING in response to a verified driver's calls to **IoCallDriver**. This option tests the driver's logic for responding to STATUS\_PENDING return values from **IoCallDriver**.

In Windows 7, the Force Pending I/O Requests option is more effective at forcing the exercising of the STATUS\_PENDING code paths in verified drivers. In earlier Windows versions, Driver Verifier forced an IRP completion to be delayed only when the first **IoCompleteRequest** for that IRP executes. This means that the effectiveness of verifying Driver1 can be reduced by the behavior of Driver2 from the same device stack. Driver2 might wait synchronously for the completion before it returns from its dispatch routine to Driver1. The forced delay of the IRP completion occurs precisely before the I/O request unwinds back into the verified driver on the completion path. This means that the STATUS\_PENDING code path of the verified driver is really exercised and the verified driver perceives a delay in the completion.

To activate Force Pending I/O Requests, you must activate both the I/O Verification and the Force Pending I/O Requests options. You can activate the Force Pending I/O Requests option for one or more drivers by using Driver Verifier manager or the command line.

To activate the Force Pending I/O Requests option at the command line

* Use a flag value of 0x210 or add 0x210 to the flag value.

This value activates I/O Verification (0x10) and Force Pending I/O Requests (0x200), as shown in the following example:

verifier /flags 0x210 /driver MyDriver.sys

The option is active after the next boot.

If you try to activate only Force Pending I/O Requests (verifier /flags 0x200), Driver Verifier automatically enables both Force Pending I/O Requests (0x200) and Enhanced I/O Verification (0x10).

You can also activate and deactivate Force Pending I/O Requests without rebooting the computer by adding the */volatile* parameter to the command, as shown in the following example:

verifier /volatile /flags 0x210 /driver MyDriver.sys

This setting is effective immediately but is lost when you shut down or reboot the computer.

To activate the Force Pending I/O Requests option by using Driver Verifier manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings **from a full list**.

3. Select I/O Verification and Force Pending I/O Requests.

If you select only Force Pending I/O Requests, Driver Verifier manager reminds you that I/O Verification is required and offers to enable it for you.

# Additional Debugging Information

In Windows 7, Driver Verifier provides the following additional information that is useful for debugging:

* There is a log with stack traces in chronological order for recent calls to **KeEnterCriticalRegion** and **KeLeaveCriticalRegion** from verified drivers. The log contents are displayed by using the **!verifier 0x200** debugger extension of the Windows Debuggers. This information can be useful for understanding scenarios in which a thread is unexpectedly running in a critical region or is trying to leave a critical region that it has left already.
* The user can display additional information from the Force Pending I/O Requests Log by using the **!verifier 0x40** debugger extension. In earlier Windows versions, the log contained just one stack trace for each IRP that Driver Verifier forced to be pending. This was the stack trace from the time when **IoCompleteRequest** was called for the first time for the forced pending IRP. Windows 7 has at least two log entries, possibly more than two, for each forced pending IRP:

Stack trace at the time when Driver Verifier picked the IRP to be forced pending. Driver Verifier chooses some of the IRPs to be forced pending when one of the verified drivers calls **IoCallDriver**.

Stack traces for each **IoCompleteRequest** call for the forced pending IRP before the completion reaches the verified driver. More than one **IoCompleteRequest** call can exist for the same IRP because one of the drivers can temporarily stop the completion from its completion routine and then resume it by calling **IoCompleteRequest** again.

* There are more valid stack traces in the IRQL Transition log. This log is displayed by using **!verifier 8**. In Windows versions earlier than Windows 7, Driver Verifier could have tried to log some of these stack traces at elevated IRQL and failed to capture the stack trace because of the high IRQL value. In Windows 7, Driver Verifier tries to capture these stack traces:

Before raising the IRQL, for example, when a verified driver calls **KeAcquireSpinLock**.

After the IRQL is lowered, when a verified driver calls **KeReleaseSpinLock**.

In this way, Driver Verifier can capture more of these IRQL transition stack traces.

* **!analyze** can triage issues that are exposed by the Enhanced I/O Verifier checks (that are part of I/O Verifier in Windows 7). In earlier Windows versions, the Enhanced I/O Verifier error reporting consisted of displaying a description of the driver defect that was detected by Driver Verifier followed by a break into debugger. Running **!analyze** after such a break does not result in meaningful triage for many of these breaks because **!analyze** cannot use the information from the error description text that appears in the debugger. In Windows 7, the meaningful information about these driver defects is saved by Driver Verifier in memory**. !analyze** can find this information and perform a much more meaningful automatic triage for many of these breaks.

# Resources

WHDC Web site:

Driver Verifier

<http://www.microsoft.com/whdc/DevTools/tools/DrvVerifier.mspx>

Debugging Tools for Windows

<http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx>

Windows Driver Foundation

<http://www.microsoft.com/whdc/driver/wdf/default.mspx>

Windows Driver Kit:

<http://www.microsoft.com/whdc/devtools/WDK/default.mspx>

Canceling IRPs

 <http://msdn.microsoft.com/en-us/library/ms795761.aspx>

Completing IRPs

 <http://msdn.microsoft.com/en-us/library/ms795821.aspx>

Driver Verifier

<http://msdn.microsoft.com/en-us/library/ms792872.aspx>

KeEnterCriticalRegion

<http://msdn.microsoft.com/en-us/library/ms801955.aspx>

KeLeaveCriticalRegion

<http://msdn.microsoft.com/en-us/library/ms801968.aspx>

Driver Verifier Options

<http://msdn.microsoft.com/en-us/library/ms792857.aspx>

Enhanced I/O Verification

<http://msdn.microsoft.com/en-us/library/aa468717.aspx>

White Papers:

Driver Verifier in Windows Vista

<http://www.microsoft.com/whdc/devtools/tools/vistaverifier.mspx>

I/O Completion/Cancellation Guidelines

[http://www.microsoft.com/whdc/driver/kernel/Iocancel.mspx](http://www.microsoft.com/whdc/driver/kernel/IOcancel.mspx)

Cancel Logic in Windows Drivers

<http://www.microsoft.com/whdc/driver/kernel/cancel_logic.mspx>