Getting Started with the Windows Driver Development Environment

Don Burn  
Microsoft MVP, Windows Driver Kit  
Windows Filesystem and Driver Consulting - windrvr.com

October 28, 2010

Abstract

This paper presents an overview of the debugging and testing tools that developersuse to create a device driver for Windows® operating systems. In particular, the paper examines ways to find and fix bugs early in development, to help you produce a high-quality device driver.

This paper provides an introduction to the Windows Driver Kit (WDK) for developers who are new to Windows driver development. Experienced developers will find the descriptions of the WDK tools a good way to start to use the new tools.

This information applies to the following operating systems:  
Windows Server® 2008 R2  
Windows 7  
Windows Server 2008  
Windows Vista®  
Windows Server 2003  
Windows XP  
Windows 2000

The current version of this paper is maintained on the Web at:   
 <http://www.microsoft.com/whdc/driver/foundation/DrvDev_Intro.mspx>

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

***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.*

*This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.*

© 2010 Microsoft Corporation. All rights reserved.

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Change |  |  |  |
| October 28, 2010 | Minor modification in Kernel Debuggers and Windows Performance Analyzer sections; added a resource for the DeviceTree tool. | | | |
| May 17, 2010 | Updated with changes to the tools, build environment, and best practices from the Windows Server 2003 Service Pack 1 Windows Driver Kit (WDK) to the Windows WDK Version 7.1. Techniques to obtain the best results with the Windows Driver Framework and the new testing capabilities of the WDK are emphasized. | | | |
| April 18, 2005 | First publication | | | |

Contents

[Introduction 4](#_Toc261869113)

[Do You Need a Driver? 4](#_Toc261869114)

[The Many Types of Drivers 5](#_Toc261869115)

[What You Need for Driver Development 7](#_Toc261869116)

[Your Test Machine 8](#_Toc261869117)

[Why You Should Find Bugs Early 9](#_Toc261869118)

[Installing the Software 10](#_Toc261869119)

[Development Environment 10](#_Toc261869120)

[Build Utility 10](#_Toc261869121)

[Build Environments 11](#_Toc261869122)

[*Sources* File 12](#_Toc261869123)

[Makefile.inc File 13](#_Toc261869124)

[Build Command Line 13](#_Toc261869125)

[Multitarget Build Projects 13](#_Toc261869126)

[Reasons to Use the Build Utility 14](#_Toc261869127)

[Using Build with Visual Studio 14](#_Toc261869128)

[Compile-Time Checking 14](#_Toc261869129)

[Compiling with /W4 /Wp64 /WX 15](#_Toc261869130)

[The C++ Compiler 16](#_Toc261869131)

[PREfast 17](#_Toc261869132)

[Running PREfast 17](#_Toc261869133)

[Using Annotations 18](#_Toc261869134)

[Suppressing False Positives 19](#_Toc261869135)

[Static Driver Verifier 19](#_Toc261869136)

[Code Preparation for SDV 20](#_Toc261869137)

[SDV Rules 20](#_Toc261869138)

[Running SDV 20](#_Toc261869139)

[Defining DEPRECATE\_DDK\_FUNCTIONS 20](#_Toc261869140)

[Using Preprocessor Tests and C\_ASSERT 21](#_Toc261869141)

[C\_ASSERT 21](#_Toc261869142)

[Conditional Tests 22](#_Toc261869143)

[32-bit and 64-bit Operating Systems 22](#_Toc261869144)

[Run ChkINF Before You Install 23](#_Toc261869145)

[Debugging and Runtime Checking 23](#_Toc261869146)

[Kernel Debuggers 24](#_Toc261869147)

[Setting Up the Debugger 24](#_Toc261869148)

[Using Driver Mapping 27](#_Toc261869149)

[WinDBG Display Windows 27](#_Toc261869150)

[Useful Commands 28](#_Toc261869151)

[Additional Test Machine Setup 29](#_Toc261869152)

[Checked Build of Windows 31](#_Toc261869153)

[Assertions 31](#_Toc261869154)

[Diagnostic Output 32](#_Toc261869155)

[Installation of the Checked Build 32](#_Toc261869156)

[Driver Verifier 33](#_Toc261869157)

[WDF Tools 34](#_Toc261869158)

[WDF Driver Verifier 34](#_Toc261869159)

[KMDF Logging 35](#_Toc261869160)

[WDF Fault Injection 35](#_Toc261869161)

[Pool Tagging 36](#_Toc261869162)

[Code Coverage 37](#_Toc261869163)

[DrvCov 37](#_Toc261869164)

[Statement Coverage 38](#_Toc261869165)

[Performance Profiling 39](#_Toc261869166)

[Kernel Logging 39](#_Toc261869167)

[Windows Performance Analyzer 40](#_Toc261869168)

[Installation Logging 41](#_Toc261869169)

[Miscellaneous Tools 41](#_Toc261869170)

[Diagnostics 42](#_Toc261869171)

[Assertions 42](#_Toc261869172)

[Debug Print Statements 44](#_Toc261869173)

[WPP Software Tracing 46](#_Toc261869174)

[Event Log 48](#_Toc261869175)

[Performance Monitoring 49](#_Toc261869176)

[Custom Dump Data 50](#_Toc261869177)

[Version Block 50](#_Toc261869178)

[Test Tools 50](#_Toc261869179)

[Device Path Exerciser 51](#_Toc261869180)

[IoSpy and IoAttack 52](#_Toc261869181)

[Plug and Play Driver Test 53](#_Toc261869182)

[PwrTest 53](#_Toc261869183)

[Windows Device Testing Framework 54](#_Toc261869184)

[Windows Logo Kit 54](#_Toc261869185)

[Summary 55](#_Toc261869186)

[Resources 56](#_Toc261869187)

# Introduction

Even for experienced developers, getting started with Windows® device drivers can be difficult. You have a new programming model to learn and a new set of tools for building and debugging. When you open the documentation, you find thousands of pages of information.

This paper presents an overview of the debugging and testing tools that developersuse to create a Windows device driver.In particular, the paper examines ways to find and fix bugs early in development to help you produce a high-quality device driver.

## Do You Need a Driver?

Before you start development of your new device and its driver, consider whether you actually need a driver. Device drivers are significant software projects that require careful development to ensure the stability of the system. Developing a driverrequires different expertise than developing a user-mode application. In addition, the cost of supporting a driver can be many times the cost of its development. Here are some questions to ask before assuming that you need a kernel-mode driver:

Can you use a Microsoft-supplied driver?

Modern peripheral buses such as USB and IEEE 1394 require devices in many classes to comply with industry standard interfaces. The host controllers for these buses also must comply with industry standard interfaces for compatibility with Windows. If your device complies with these standards, it can use the drivers that Microsoft provides.

Is the driver for a standard device class that has a unique hardware design?

If your device is in a standard device class—such as a storage device—but has a unique hardware design, you need a driver. Similarly, if your device is accessed through a Win32® API, a driver is required.

**Can the driver use the user-mode driver framework (UMDF)?**

Drivers that communicate by using a protocol such as USB can often be moved to user space. If you do this,a failure of the driver impacts only the device and avoids a system crash.Even if your device must be programmed in the kernel, consider creating a minimal kernel driver to support a UMDF driver. UMDF is a rapidly evolving technology.Even if you previously excluded developing your driver in usermode,investigate the capabilities of the current UMDF before you develop a kernel-mode driver.

* **Do only a few applications access your device through a user-mode API?**

A driver might be unnecessary if your device connects to the system through a link to an existing driver that has a user-mode API and if only a few applications use the device. A device that is connected to the system by a serial link or an Ethernet protocol is unlikely to requirea driver. Additionally, devices that are connected to a storage bus such as Small Computer System Interface (SCSI) or ATA Packet Interface (ATAPI) can use the pass-through capabilities in the bus controller device driver to support sending and receiving requests from devices that are connected to the bus.

Is your driver a software-only driver?

A software-only kernel-mode driver is another type of driver that is sometimes unnecessary. This type of driver can often be replaced by a user-mode service that runs under the system account, a UMDF driver, or even a user-mode application.

* Is your driver for a USB-connected device that is used exclusively by one application?

A driver is unnecessary if your USB-connected device does not use isochronous endpoints and is used exclusively by one application.In this situation, you can use *winusb.sys* and the user-mode WinUSB APIs to communicate with the device.

Can you move some development to a user-mode DLL?

Finally, if you conclude that your device requires a driver, consider using a user-mode DLL to reduce the amount of kernel development that is required for your project.

## The Many Types of Drivers

The Windows Driver Kit (WDK) and the kernel interfaces that it supports are the programming model for third-party developers for the Windows kernel.New developers can be confused because a driver can be a module that is not associated with any hardware.This section describes how to choose a driver type for your device, although a physical device is not necessary for a driver.

Windows versions earlier than Windows 95 had a simple driver model in which a driver queried the bus to determine the devices that it supported and the required resources for each device.In Windows 95 and Windows 2000, the need to enable a computer system to recognize and adapt to hardware configuration changes such as plugging in a USB device led to the addition of Plug and Play and power management to the driver model.With the advent of Plug and Play, the earlier driver model has become known as the *legacy model*, which is no longer supported if hardware resources are used.

The major change in the Plug and Play driver model is that the operating system notifies the driver of the presence of a device and provides the resource data such as interrupts, memory, and I/O ports to the driver. The operating system can also remove or reassign resources for the device.

Windows has several types of device drivers, each with its own interfaces and capabilities. The device class and the particular bus type that the device uses to connect to the system are important considerations when you choose the correct driver type.Many device classes (such as storage and networking) have specific interface requirements for the layers above your driver. For these device classes, you are likely constrained to the driver type for that class of device.Beyond the device class, the type of bus impacts the choice. For instance, it is possible to move the driver into user space by using UMDF for buses such as USB. Another example is an API extension to access some kernel feature to support an application. In this example, you should consider a legacy model if no hardware is involved.

Understanding the different driver types is important for several reasons:

* The kernel programming interfaces can vary from model to model.
* The rules for what the driver can do vary by type.
* If you ask someone for help, that individual needs toknow the type of driver that you are developing.

Windows device drivers can be classified in many ways. One way is that drivers can take on one of the following roles in a driver stack:

Function drivers.

A function driver provides the core functionality of the driver stack.It is what people traditionally think of as a device driver. The function driver’s upper edge defines the interface that the remainder of the drivers in the stack—and typically the system and applications—use. For a physical device, the function driver’s lower edge manages communication with the device. For Plug and Play–enumerated devices, the function driver manages the Plug and Play and power states of the device.

Filter drivers.

A filter driver layers above or below an existing driver to provide special services or change the behavior of a standard driver. An example of a filter driver is an encryption filter that encrypts write operations on the way to the function driver and decrypts the data that is read from the function driver.Filter drivers typically follow the rules of the type of driver that they filter.

Bus drivers.

A bus driver enumerates devices on a bus and arbitrates access to the bus. Bus drivers are Plug and Play drivers and must support special requests beyond what other Windows drivers do. Many bus drivers are provided by the system, but third-party bus drivers are common.An example of a third-party bus driver is a driver that creates multiple devices from a single piece of hardware.

Another way to consider drivers is their programming model. The first two models in the following list are for various devices, whereas the subsequent models are for specific needs:

Windows Driver Foundation (WDF).

As the Windows Driver Model (WDM) grew, it became more difficult to program. With the release of the Windows Vista® WDK,we released a new driver model that relieved many of the problems of driver development. The WDF modelhas two implementations: kernel-mode driver framework (KMDF) and user-mode driver framework (UMDF). KMDF provides a replacement model for WDM while allowing you to access WDM features as you need them. UMDF lets you move the drivers for some categories of devices to the user space.

Windows Driver Model (WDM).

WDM is the traditional model of device drivers for Windows. It was originally the common subset between Windows 98 and Windows NT as defined in *wdm.h*.Now it means driversthat are written to the native kernel support routines.

Legacy driver.

A legacy driver is similar toWDM, but without support for Plug and Play and power management.

File system drivers.

File systems are in a special category of drivers and are some of the most difficult drivers to write. These drivers monitor files on a device or provide access to files across the network. File system minifilters are the recommended model for file system filters, whereas all other drivers are an extended WDM model.

Storage miniport drivers.

Drivers for storage controllers such as SCSI and Integrated Device Electronics (IDE) have their own programming interface. The three major types of storage miniports are SCSI port, Storport, and ATA port.

Network drivers.

Like storage drivers, network drivers have their own programming interfaces. The major interface models for networking are Network Driver Interface Standard (NDIS) and the Windows Filtering Platform (WFP).For a network adapter, an NDIS driver can use KMDF as part of the miniport model of NDIS.

Printer drivers.

Printer drivers are a special category of drivers that run in user mode and do not conform to the programming model for other drivers. The common models for printer drivers are Universal Printer Driver (Unidrv), Postscript (PScript), and XML Paper Specification (XPS) drivers.

Graphics drivers.

Graphics drivers, another special category of drivers, have their own rules and programming interface. They are some of the most complex projects in Windowsand conform to the Windows Display Driver Model (WDDM).

Kernel-streaming drivers.

This is another unique category of driver. Like several other categories, kernel streaming (KS) drivers have unique interfaces for programming. AVStream and Port Class Adapter are the common models for KS.

The WDK has samples for all thesedriver types, which letsyou get started with a strong code base.

## What You Need for Driver Development

For driver development, you need a system for development and a system on which to debug and test your driver. Any reasonably powerfuldesktop system can provide the hardware for development. A good set of minimal requirements are the system requirements for the current version of Microsoft® Visual Studio® plus the ability to run a virtualization system for debugging as described in “Your Test Machine,” which is the following section in this paper. The only special systemrequirement is a free serial or IEEE 1394 port for the debugger. Sometimes a USB port works, but most developers want tohave a 1394 or serial port. What you need for the test machine is discussed in the following section of this paper. This section concentrates on the development system software.

The WDK provides the build environment for developing a device driver for Windows and includes the approved build tools, compiler, and linker. The kit also includes the WinDbg debugger for debugging drivers. The WDK also has thousands of pages of documentation about developing drivers, a large collection of tools, and numerous sample drivers. In fact, the only required software to build a driver other thanthe WDK is a good editing tool.

A common misconception is that you need the WDKthat is named for the particular version of Windows that your driver targets. In fact, you need only the newestWDK, which has the latest tools, samples, and documentation for how to develop drivers. Beginning with Windows 7, the WDK has a version number to identify it. Before Windows 7, the kits werenamed for the latest operating system at the time of their release. This means that the Windows 7WDK is not just for creating drivers for that version of Windows, but is the best choice for creating drivers for all versions of Windows later than Windows XP. If you have a requirement for Windows 2000, use the Windows Server 2008 WDK.

You should consider purchasing a code coverage tool. This tool can help in debugging and validatingthat you are testing your driver well. A full discussion of code coverage is provided in “Code Coverage” later in this paper.

Finally, for most Windows versions, your company needs a code-signing certificate so that you can digitally sign the 64-bit versions of your drivers. If your company does not have a certificate, start the process early so that you have time to order it and to create corporate procedures for how to useit. For more information, see “Driver Signing Requirements for Windows” on the WHDC website.

## Your Test Machine

Many developers and their managers are surprised that a separate test machine is required, but this requirement has many good reasons. Debugging an operating system is a challenge because the debugger itself affects normal operation. To limit this impact, Windows requires a separate machine for use with the WinDbg debugger. A separate machine is needed for several other reasons:

* Although your development machine typically runs a desktop version of Windows, for testing you need several Windows versions, from server to desktop. In addition, both the checked and free builds(as described in“Debugging and Runtime Checking”later in the paper) should be available on your test system.
* Crashes are likely whileyou develop a driver. At best, a crash requires rebooting the system to fix the bug and then rebooting the system again to install and run the driver. This is a slow process. At worst, you can corrupt the disk and must rebuild your system and recover your work.

On the test system, you should install multiple versions of Windows. Even if you are targeting a particular Windows version, the latest version gives you the best validation tools. One of the versions that run on the test system should be the checked build of Windows, which is one of the most powerful tools for debugging and testing your driver. How to use the checked build to improve your drivers is discussed in various sections in this paper. The checked build is available only as part of the Microsoft Developer Network (MSDN®). For availability of the checked build and other developer resources on MSDN, see “Resources.”The best way to install multiple versions of Windows is to have a disk that is large enough for a separate partition to hold each version of Windows. If you install the versions in order from oldest to latest, the Windows installer can support all of them.

For some classes of drivers, a virtual machine (VM) can work for initial development and testing. Two examples of products that can support a virtual environment for debugging are the Microsoft Hyper-V and Virtual PC. The exact classes of drivers that you can develop partially depend on the VM. Typically, you should still test the driver on real hardware as part of development.

Now that you know the reasons for having a separate test system, be sure to obtain a good system. If you have a single test machine, it should have the following components:

* At a minimum, a serial, an IEEE 1394, or sometimes a USB port for the debugger.
* A large number of coresandprocessors. Ideally, the test machine should have more processing units than most systems on which the driver will run. Good testing with a large number of processors is critical to finding multithreading bugs.
* A 64-bit system with more than 4 GB of main memory. Several bugs with direct memory access (DMA) and pointers occur only if your machine has more than 4 GB. The best way to test for these cases is with a large system.
* Pluggable disks,which make it easy to test against multiple versions of Windows.

If multiple systems are available, you should have a variety of performance capabilities for testing.For example, you can exert more stress on the driver and help find bugs that low-resource conditions causewhen you test on a system that is less powerful than most systems on which the driver will run. At a minimum, use the options in “Modifying Boot Settings of the Test Machine” later in this paper.

## Why You Should Find Bugs Early

Driver debugging is difficult, so if you find and fix bugs early you can reduce development time. This is not the current practice for many developers. The remainder of this paper describes the environment and tools you should use to build Windows device drivers. Much of the discussion involves enabling as many checks as possible.

If you find a bug at compilation time, it takes seconds to fix the code and resubmit the module for compilation. If the bug goes undetected until the runtime verification tools find it, it can take at least five minutes just to rebuild and install the driver to fix the bug. If the runtime checks do not find the bug, the five minutes stretches to hours or days.

As soon as quality assurance (QA) is involved, the costs escalate. QA must find the bug, try to reproduce it, and put it in a bug report. Then management looks at the bug report and discusses it with the developer. Finally, the developer tries to reproduce the problem and then debug it. Overall, this process can take many days. The process takes even longer if a customer reports the bug.Although the effort might not increase as rapidly, the time until the customer receives the bug fix still grows enormously.

Therefore, the little extra time it takes to enable the warning messages and fix the problems that they report is inconsequential when compared to the time it can save by finding even a single bug early in the development cycle.

## Installing the Software

Before you install the WDK and the debugger, you must know several things:

* Not all tools in the WDK work if the directory path for the WDK or the path to your driver code-base contains a space. Do not install the WDK at some location such as “C:\Program Files\WDK”. If the path contains a space, you do not receive a clean error message. Instead, the build of your driver might fail in mysterious ways.
* At a minimum, load the full build environment from the WDK. By doing this, you receive the tools that are required to build a driver, samples from which to start your driver development, additional tools (many of which are discussed in this paper), and the documentation. Although some developers use the documentation on the Web, it is much easier to find what you are looking for when you search only documentation that is related to the WDK.
* The WDK includes debugging tools for Windows,part of which is the debugger that you need to debug your drivers.
* Although the WDK includes the Device Simulation Framework (DSF),you should install DSF on test machines and not on development systems. The framework is mainly for testing certain classes of USB drivers.
* The WDK provides the Windows Device Testing Framework (WDTF).Install WDTF on the development system to provide a programming framework for creating tests for your driver. In addition, installing it with the WDK includes a directory that you can then move to the test system to install the framework and example scripts for testing.

# Development Environment

This section explores the development environment that is required to compile and link Windows device drivers. The Build utility controls the actual process of compiling and linking the driver. The remainder of this section provides an overview of the Build utility and the input file for the tool. The end of the section looks at an approach to running Build from the Visual Studio integrated development environment.

## Build Utility

You use the Build utility to create a driver in the WDK. This utility invokes the correct tools with the appropriate options to build your driver. It is a wrapper around the Microsoft NMAKE utility, so it handles issues such as checking dependencies to ensure that the correct files are rebuilt after a change.

If you have not worked with Build before, you might want to try to build one of the samples from the WDK:

1. Copy the files under src\general\<sample> (where <sample> is the particular sample’s name) to a test directory.

2. Click **Start**>**Programs**>**Windows Driver Kits**>**WDK <version>**>**Build Environments**.

3. Choose one of the build environments. The build environment opens a command window.

4. Change the directory to your test directory.

5. Type build to create the driver.

For most drivers, setting up the Build tool environment is simple:

1. Copy the standard makefile from a WDK sample project.

2. Create a sources file in the directory where your code resides.

Note that you should not change makefile. All commands are set up by the sources file or makefile.inc, as discussed later in this section. The following is an extremely simple sources file:

TARGETNAME=test

TARGETTYPE=DRIVER

SOURCES=test.c

This file sets three macros that instruct the Build utility to create a driver that is named test.sys from the test.csource file.

## Build Environments

Earlier in this paper we told you to choose one of the build environments. The current WDK has build environments for the four targeted versions of Windows (Windows XP, Windows Server 2003, Windows Vista and Windows Server 2008, and Windows 7). Under each version are environments for checked and free builds and for each CPU target that is supported for that version:

* The checked build creates a driver that has debugging enabled. Many of the optimizations in the compiler are disabled, and conditional code for debugging is included. While you work to get your driver functional, build your driver with the checked build.
* The free build creates a production driver in which the code is optimized and debug code is disabled. Use the production driver for performance testing and to ship to customers. Do not mistakenly ship the checked build driver—your customers deserve the best performing driver.

The Build utility names several files and directories that are based on the chosen environment. These names include the *Type***\_***Version***\_***Cpu* construct, as shown in the following table.

|  |  |
| --- | --- |
| *Type* | “Fre” for the free build and “Chk” for the checked build. |
| *Version* | “wxp” for Windows XP, “wnet” for Windows Server 2003, “wlh” for Windows Vista and Windows Server 2008, or “win7” for Windows 7 and Windows Server 2008 R2. |
| *Cpu* | “x86”, “amd64”, or “ia64”, depending on the processor. |

As stated earlier, building and testing for the latest version of Windows provides additional checks for your driver. Building for the earliest version of Windows that you plan to support creates a binary of your driver that is compatible with all versions of Windows from that version forward.

As an example of this,assume that you are developing a driver that must support Windows XP, Windows Vista, and Windows 7, with Windows XP being your largest current customer base. In this case, you should use the “Windows XP” build environments for building your driver binary, so that you can create a single binary that is compatible with Windows XP as well as Windows Vista and Windows 7. For testing, you should not focus on Windows XP only just because that is your largest customer base. Instead, you should also spend considerable time on Windows 7 because additional checks and validationpoints are incorporated into the latest version of Windows.

## Sources File

The example at the beginning of this section had a very simple sources file, but the recommended file is more complex. The sources file has many macros and environment variables, all of which are documented in the WDK. The following table lists items for the sources file that is used in this paper.

|  |  |
| --- | --- |
| C\_DEFINES | Specifies switches to pass to the C compiler and the resource compiler. |
| INCLUDES | Specifies directories in addition to the source directory to search for Include files. |
| MSC\_WARNING\_LEVEL | Sets the warning level for the C compiler. The default is /W3. Recommended is /W4/Wp64 /WX. For details, see "Compile Time Checking" later in this paper. |
| NTTARGETFILE0 | Causes the inclusion in the build process of *makefile.inc.* Typically used to support Windows Management Instrumentation (WMI). |
| SOURCES | Specifies the files to be compiled. |
| TARGETNAME | Specifies the name of the binary to be built. This name does not include the file name extension. |
| TARGETLIBS | Specifies additional libraries and object files to link to the product. |
| TARGETPATH | Specifies the destination directory for all build products. In almost every case, you should set TARGETPATH=obj. |
| TARGETTYPE | Specifies the type of product that is being built. Common values are DRIVER (kernel-mode driver), PROGRAM (user-mode program), and DRIVER\_LIBRARY (kernel-mode library) |
| USER\_C\_FLAGS | Specifies switches to pass only to the C compiler. |

For several capabilities that are discussed later in this paper, it is valuable to know about the following construct in the sources file. Sources has conditional tests, such as the following example:

!if !$(FREEBUILD)

...

!endif

This sample specifies that the actions between !if and !endif occur only in the checked build of the driver.

## Makefile.inc File

Makefile.inc provides additional information to the Build utility. This file letsyou add commands and dependencies to *makefile*, typically for preprocessing and postprocessing of files that are related to the project. One common use in driver development is to compile a Managed Object Format (MOF) file that is required for WMI and produce an Include file that contains data structures for the driver.

## Build Command Line

As part of finding bugs early, consider some options for the Build command line. At a minimum, consider adding –bew to the build command, as shown in the following table.

|  |  |
| --- | --- |
| **b** | Enables full error messages. |
| e | Generates the log, error, and warning files that can help you when an error occurs in the Build utility. |
| w | Displays the warning messages to the command window, so you do not miss them. |

Periodically, and before you ship your driver, you should use the –c option to rebuild all files. The –c option deletes all object files.

The Build utility has many options, which are all documented in the WDK. Ata minimum, consider using the ones that are mentioned here to validate your work.

## Multitarget Build Projects

Building most driver projects is relatively simple.You put the files that you need in a single directory and build the driver. Many developers encounter difficulties when the project becomes more complex.

If your project has multiple components, each component should be in its own directory with a *dirs* file to identify the individual directories. The *dirs* file typically contains a single definition of the following form:

DIRS=dir1 dir2 dir3

dir1, dir2, and dir3 are subdirectories for components of the directory that contain the *dirs* file. These components can contain subcomponents.In fact, you can build all examples in the WDK as a single project. The simple model of a driver and perhaps several support components such as an application and a DLL represent most projects.

If you have a multicomponent project, sharing files is the most likely problem. The Build utility allows you to use source files from the current directory, its parent directory, and the platform-specific subdirectory. The common approach places the shared files in the parent directory and the unique files in the current directory as shown in the following table.

|  |  |
| --- | --- |
| Current directory | Contains all files for the project except those source files that must be shared with other components. |
| Parent directory | Contains source files that are shared between two or more of the subdirectories of this directory. |
| Platform-specific directory | Contains any files that are specific to this particular platform. |

Do not overlook the limit of thesedirectories. The SOURCES= line that specifies the files to compile can reference only these directories. Some companies develop rules for source trees that are incompatible with the requirements of this Build tool. This incompatibility can have a major detrimental impact on a driver project. If you must break the source files into multiple components, the common approach is to use a static library for each subcomponent and then link the libraries when you build the driver.

## Reasonsto Use the Build Utility

The Build utility is the only utility that you should use to compile and link a device driver. It is a wrapper program for NMAKE (the Windows make program) and is provided with the WDK to control compiling and linking a devicedriver and related code. By using Build, you ensure that the driver follows the conventions and settings that Windows requires. Build handles all target-dependent problems, including different paths to compilers and Include files for specific platforms. Additionally, Build supports different target directories that are based on both the target platform and whether debugging is enabled.

Using tools other than the Build utility creates three types of problems:

* First, many WDK tools use the Build environment. If you do not use Build, you cannot use these tools.
* Second, if you believe you have found a bug in the WDK, Microsoft support requires code that is built with the Build utility to give you customer support.
* Finally, if you do not use the Build utility, you can easily have the wrong settings for Windows drivers and create subtle bugs that can take a long time to find.

## Using Build with Visual Studio

One of the main reasons that developers try to bypass Build is to use Visual Studio. You canuse Visual Studio as an editor and integrated development environment, but you should have Visual Studio invoke the Build utility by using a batch command file.

Fortunately, you do not needto worry about the details yourself because two versions of a batch command that is called DDKbuild are free to download. For various sources of this batch command, see “Resources.”

If you choose to use this approach, you should test that your project can still be built in the WDK build environments. This testing guarantees that nothing unusual occurs because of the batch command.

# Compile-Time Checking

As stated earlier in this paper, a principle of good software engineering is to find bugs early. Studies have indicated that the cost of finding and fixing a defect increases ten times for every step of the development process. So a bug found at compile time can be fixed at 1 percent of the cost of a bug that makes it to testing.

Developers should follow careful software engineering principles, but in the pressure of product development schedules, this can be difficult to do. Additionally, many developers do not have formal training inthese best-practice methodologies. Fortunately, the Windows WDK provides several capabilities to find bugs before running the driver. Using these tools does not require the developer to have formal training to find bugs early.

You should note that the same problem can be found by more than one of the techniques that are described in this section. You should resolve all reported problems from one tool before running the next tool. And you should use all techniques because each can find unique problems.

Finally, several techniques can create *false positives*, where a warning or reported error can never happen. These false positives occur because error-detecting tools must be thorough when reporting any possible errors. It is still important for you to resolve all warnings. Fortunately, eliminating most of the warnings is easy. View the effort to suppress the warnings as a “small code review.” You must check whether the warning is valid anyway, so look at the code around the warning and validate that no other errors are present. These checks cause you to look at your code in a different manner than you typically would look at it, so use this fresh perspective to look for problems.

## Compiling with /W4/Wp64 /WX

A simple way to improve your code is to compile with warnings that are enabled and treated as errors. The /W4compiler option enables most warnings. The /WX option treats warnings as errors. The Include files from the WDK compile cleanly with the **/W4** option, but some sample programs still produce warnings if **/W4** is enabled. In addition to **/W4**,you should use **/Wp64** to find portability problems with 64-bit code. You should note that**/Wp64** produces a warning about the feature that is being deprecated in the future. You can ignore thiswarning. To enable the additional warnings, use the following line in a *sources* file:

MSC\_WARNING\_LEVEL=/W4 /Wp64 /WX

These warnings add simple checks that can indicate potential problems. The following are some of the common warnings that are likely to appear for your driver:

C4057 'operator' : 'identifier1'indirection to slightly different base types from 'identifier2'

Typically, this warning appears because of two similar but different types, such as a signed versus an unsigned value or a short versus a long value. To fix this, change the base type where possible to the correct form or use a C cast to convert the item.

C4100 '*identifier*' : unreferenced formal parameter

If this warning occurs in a function that you designed, determine why you passed this parameter. This warning occurs mainly where the WDK dictates the function prototype. In this case, use the UNREFERENCED\_PARAMETER macro to eliminate the error. Be careful! If your driver later begins to use the parameter, be sure to delete the UNREFERENCED\_PARAMETER macro for that parameter.

C4101 '*identifier*' : unreferenced local variable

The simple solution here is to delete the unused variable. Before you do so, determine why it was there. Often an unused variable indicates that something was forgotten in the function, so check your code and design before you delete the variable.

C4244 '*variable*' : conversion from '*type*' to '*type*', possible loss of data

This warning occurs when you convert an integer value to a smaller type. This can reflect a real error such as forcing a 64-bit value to 32 bits. Runtime checks can be enabled for this problem.So, whenyou fix this error, consider ANDing the value with a mask to a legal size and then casting the value to the correct type.

## The C++ Compiler

C++ has been described as “a better C.” This statement is not a recommendation to use C++ language features in the kernel. Using C++ in the kernel presents several challenges, so unless you are comfortable with Windows driver development you should avoid using C++. For information to help you make your own decision, see“C++ for Kernel Mode Drivers: Pros and Cons”on the WHDC website. One problem that is not mentioned in the paper is that several Microsoft tools for driver writers either do not support C++ or support a subset of the language.

Be aware that if you mix C and C++ in your driver, the Build tool does not correctly handle two files with the same name but differing extensions such as *test.c* and *test.cpp*. In fact, having two such similar files in the same directory causes problems even if they are not both used in the driver.

Although using the full C++ language in the kernel can cause problems, using the C++ compiler can help you identify potential improvements in your driver because the C++ compiler does more validating and checking.The challenge is to be sure that you are still programming in C while compiling with C++.

One approach is to use the compiler's /TP switch to treat files with a .c extension as C++ code. Using this option in the sources file only for the checked build lets you discover any C++ constructs in your code, as shown in the following example:

!if !$(FREEBUILD)

USER\_C\_FLAGS= /TP

!endif

This example builds a driver by using C++ for the checked build. If you do this, be sure to change your DriverEntry routine also, as shown in the following example:

#ifdef \_\_cplusplus

extern "C"

#endif

NTSTATUS DriverEntry(   
    IN PDRIVER\_OBJECT  DriverObject,   
    IN PUNICODE\_STRING  RegistryPath);

You might also be required to use extern "C" to wrap the Include file for Windows preprocessor (WPP) software tracing (which is described in the “WPP Software Tracing” section later in this paper). This technique requires building both the checked and the free build of the driver. If you use Visual Studio with ddkbuild, use the batch build command to build both versions simultaneously.

Most of the errors that are found by using the C++ compiler involve type checks and certain constructs that are not allowed in C++. As with /W4 /Wp64, resolvingthese warnings can reduce the number of actual errors in your driver.

## PREfast

PREfast is a tool that wedeveloped to find several problems that are difficultfor a compiler to locate. To locate problems, it performs a static analysis of each function in your code. Unlike Static Driver Verifier (which is described in “Static Driver Verifier” later in this paper), it does not perform anentire program analysis.Because PREfast runs at the function level, it can perform its analysis quickly. TheWDK provides a version of PREfast that has driver-specific rules that every driver writer should use. This version of PREfast can detect several common problems for drivers through explicit knowledge of kernel-mode functions and structures. The following are some of the types of errors that PREfast finds:

Variable problems.

This includes uninitialized variables, NULL pointer dereferences, and even out-of-range parameters.

Leaking memory.

This is commonly things such as forgetting to free allocated memory.

Interrupt request level (IRQL) problems

This includes a wrong IRQL, raising an IRQL without lowering it, and other similar problems.

Accessing private structures.

Some WDK structures contain fields that the driver should not directly access. PREfast can find these problems.

### Running PREfast

PREfasthas become such a powerful tool with the Windows 7 WDK thatwe added a new tool that is called Microsoft Auto Code Review (OACR). OACR detects when you have built a driver and runsPREfastautomatically. This is the simplest way to use PREfast. When you build your driver, a small icon appears in the tool tray of the task bar. Click the icon, and the PREfastgraphical user interface (GUI) defect viewer displays all warnings that the tool finds.

If you want tighter control over PREfastthan what OACR provides, you can manually invoke PREfastas part of your Build utility command line as shown in the following example:

prefast /list build –cefbw

This line builds the driver, runs PREfaston the sources, and displays the defect log on the console. For a description of all PREfastcommand options and for commands to disable OACR if you want to, refer to the WDK documentation.

### Using Annotations

To get the most from PREfast, you must provide information about the code that is being processed. To learn how to add annotations to your code, see “PREfast Annotations” on the WHDC website. The following list shows a few common annotations:

\_\_in

This parameter is for input, so you should initialize it.

\_\_out

This parameter is for output.

\_\_inout

This parameter is for input and output.

\_opt

This parameter can be added to one of the preceding three annotations. If the parameter is a pointer, it can be NULL.

\_bcount(size)

Typically added to one of the preceding annotations, such as **\_\_in\_bcount(*size)****,*this parameter indicates the size in bytes of the parameter.Size can be another parameter or a constant.

\_ecount(size)

Used like **\_bcount**, this parameter indicates the number of elements for a parameter.

\_\_checkReturn

This parameter is added to indicate that the driver must check the return value of the function.

\_\_drv\_maxIRQL(v)

This parameter indicates that the function can run only at an IRQL level less than or equal to “v”.

The following example shows a simple function that is being annotated:

\_\_drv\_maxIRQL(PASSIVE\_LEVEL)   
\_\_checkReturn   
NTSTATUS TestFunction(

\_\_in\_bcount( len ) PCHAR buf, \_\_in ULONG len );

This example indicates that the function has the following rules:

* The maximum IRQL to call the function is PASSIVE\_LEVEL.
* The results of the function must be checked.
* The first parameter is an input buffer whose length is specified by the second parameter.

The WDK documentation has a complete list of the annotations. In addition, see “PREfast Step-by-Step” and “PREfast Annotations” on the WHDC website for more information.

### Suppressing False Positives

When PREfast is aggressive, it can produce several warnings that falsely indicate a problem. Some developers just ignore these, but there might be a real problem hiding in the errors. PREfastprovides several ways to suppress these warnings. You can use these so that only new problems are reported as you develop your driver and you can determine early how to fix or suppress them. The following are several approaches to suppress errors:

Minor coding changes.

Several minor coding changes can significantly reduce the number of warnings. Two common examples are initializing variables when they are declared and explicitly testing pointers for NULL before you use them. The PREfast papers in “Resources” list most examples.

Annotations.

You can often use annotations to suppress false positives.For example,some annotations can indicate that the condition that caused the warning was expected. Also, good annotations allow the tool to better predict behavior and reduce false positives.

#pragma to suppress errors.

When minor coding changes and annotations donot work, use #pragma to disable the warning. Do this only for the line that has the false positive. Disabling the warning for the entire source file can hide valid problems that PREfastfinds.

PREfastfilters.

This is the least recommended approach to removing false positives. When PREfastwas new, filters were provided to suppress certain warnings, and in earlier versions of the tool these were enabled by default. Unfortunately, often these hidevalid bugs, so we now recommend that you do not use filtering.

## Static Driver Verifier

Static Driver Verifier (SDV) is a tool that finds problems in a driver’s interactions with the operating system. SDV can check legacy, WDM, KMDF, and NDIS drivers. SDV examines the entire driver and in particular how a path through the driver interacts with kernel services. Unlike PREfast, which is designed to run on every build, SDV can take a long time to run. In addition,you should run SDV onlyafter you have a clean compile and preferably after you have resolved all PREfastwarnings.

By looking at the entire request path, SDV can find bugs in handling the request such as completing twice or not freeing locks that are difficult to find.Many of these problems are rarebecause of race conditions or code that is executed under rare circumstances, which makes them extremely difficult to find through regular testing.

### Code Preparation for SDV

For SDV to work correctly, it requires a hint in the driver’s code. The various entry points and callbacks must be identified by their role type. The role type is a standard C language type, as shown in the following example:

DRIVER\_INITIALIZE DriverEntry;

NTSTATUS DriverEntry(

IN PDRIVER\_OBJECT DriverObject,

IN PUNICODE\_STRING RegistryPath )

This example indicates that**DriverEntry** is the initialization routine for the device driver. You should run PREfastover the driver and resolve any warning in the code because PREfastchecks that role types are properly defined for the driver.

### SDV Rules

SDV is rule-based. Each rule is checked separately against the driver code to see if a problem exists. Many rules are available, and the rules are associated with the type of driver and then further classified by what they test. The rules are in the WDK documentation and are individual files under the *rules* directory of the SDV tool. Currently, the simplest approach is to periodically run SDV against all rules and then run SDV against specific rules that failed to determine if errors were resolved.

### Running SDV

SDV must run in the BUILD command window. The first time you run SDV on a project (or if you change the names of common entry points), you must complete the following steps:

1. **staticdv /scan**Issue this command to identify the entry points.

2. **Edit Sdv-map.h**Check the *sdv-map.h* file that the previous command created.If it is correct, change the first line to: “// Approved=true”.

After you have set up the driver, you can perform the following steps to run SDV (running SDV with all steps can be a lengthy process):

1. **staticdv /clean**This command deletes the previous data that SDV created.

2. **staticdv /rule:**\*  
This command runs SDV with all rules.

3. **staticdv /view**This command shows the errors that were found.

The full capabilities of SDV are documented in the WDK. The tool requires some effort, but it is well worth the work when it finds a bug for you.

## DefiningDEPRECATE\_DDK\_FUNCTIONS

The WDK has evolved over the years. As a result, several functions are no longer recommended and supported. The Windows logotest specifications define a set of functions that should no longer be used. The WDK provides a way to check for functions that should not be used by defining DEPRECATE\_DDK\_FUNCTIONS in sources as part of a C\_DEFINES line. If you define DEPRECATE\_DDK\_FUNCTIONS, these functions generate errors at compile time. You should note that PREfast also flagsseveralof these functions.Both using PREfast and defining DEPRECATE\_DDK\_FUNCTIONS provide the best coverage.

However, DEPRECATE\_DDK\_FUNCTIONS does not flag every function in the WDK that you should not use. Sometimes, these functions are correct for an earlier version of Windows but have been supplemented by a new routine. For this reason, you should check the WDK documentation when you use a function for the first time or when a newer WDK becomes available, to see if you are using the best routine.

If you encounter a situation in which a function is obsolete in later versions, you have two choices:

* Typically, the function is supported even after it is obsolete. One approach is to use the function for all versions of Windows, even though it is obsolete in some versions. The problem is that the function might be removed later and your driver would stop working.
* A better approach is to use MmGetSystemRoutineAddress to check for the presence of the functions. Be sure to check for both the old and the new function, and fail with an error if neither function is present. Your driver might last a long time, and both functions could be obsolete someday.

## Using Preprocessor Tests and C\_ASSERT

Many developers know about runtime assertions, but do not use them at compile time. Many conditions can be tested at compile time. As discussed earlier in this paper, finding bugs early saves time.

You can approach compile-time assertions in two ways: C\_ASSERT and conditional tests. Both approaches work.The important thing is to test at compile time.

### C\_ASSERT

One of the most overlooked capabilities in the WDK is the C\_ASSERT macro, which checks assertions at compiletime. The expression in C\_ASSERT must be determinable at compile time. The checks can include testing use of the sizeof operator on type declarations and using compile-time constants. An assertion failure results in *error C2118: negative subscript*. The following are two common examples of C\_ASSERT:

typedef union {

// Warning: Size of elements must be the same

TYPE\_1 element1;

TYPE\_2 element2;

} UNION\_1;

C\_ASSERT ( sizeof(TYPE\_1) == sizeof(TYPE\_2) );

// Warning: Tables need the same number of entries

PCHAR table1[] = {“a”, ... };

PCHAR table2[] = {“AA”, ... };

C\_ASSERT( RTL\_NUMBER\_OF(table1) == RTL\_NUMBER\_OF(table2));

Typically, drivers have a comment that notes that TYPE\_1 and TYPE\_2 must be the same size or that two arrays must have the same number of elements. These comments are easy to see, and you should fail the compilation if the requirements are not met.

### Conditional Tests

One problem with compile-time assertions is that you receive the negative subscript error but no useful data until you look at the code. So you might want to consider changing your C\_ASSERTs that use only preprocessor data to do conditional tests. The compiler provides a #error statement that generates an error. The error is reported as a message on the line following the #error. The following are two common examples:

#define BLK\_SHIFT 9

#define BLK\_SIZE 512 // Must be 1 << BLK\_SHIFT

#if !(BLK\_SIZE = (1 << BLK\_SHIFT))

#error !(BLK\_SIZE == (1 << BLK\_SHIFT))

#endif

// Need version 5 or module should be changed

#include “driver\_include.h”

#if !(DRIVER\_INCLUDE\_VERSION == DRIVER\_INCLUDE\_V5)

#error !(DDRIVER\_INCLUDE\_VERSION == DRIVER\_INCLUDE\_V5)

#endif

The advantage to this approach is that the error message you see matches the assertion so that you do not need to check the code.

## 32-bit and 64-bit Operating Systems

All the precedingapproaches help you find bugs at compile time, but only for the targets for which your driver is compiled. Compiling for both *x*86 and *x*64 can find problems in your code. In particular, compiling for 64-bit operating systems can find problems in the following three major areas:

Pointer issues.

The most common problems when you migrate to a 64-bit operating system are pointer problems. Example problems include errors in pointer arithmetic, trying to store a pointer in a 32-bit value, or having a structure change in size because of the pointers that it contains.

Inline assembly language.

You should not use inline assembly language (or assembly language at all) in your driver, but sometimes this occurs. You might not even realize it and might use an Include file or a piece of code from another developer who used assembly language. This is the time to findand fix this code.

Compiler differences.

Several subtle differences in the compiler are described in the WDK under “Code Compatibility for *x*64.” Although it is unlikely that you will encounter these differences, it is possible.

Be aware that you cannot find all compatibility problems for the driver without testing on both platforms. Compiling cleanly for both platforms means you have taken a good first step.

## Run ChkINF Before You Install

ChkINF is a commonly overlooked tool. Some developers view it as a Windows logo test rather than a development tool. ChkINF is one of the best ways to ensure that your driver installation works the first time. No INF file is ready to use until ChkINF validates it and you eliminate all possible errors and warnings.

ChkINF resides in the \tools directory of the WDK. The WDK also documents all command-line arguments for the tool. The following is a common approach to running ChkINF:

ChkINF test.inf /B

This command validates *test.inf* and invokes the browser to display an HTML file that has the results. The report consists of a section with errors and warnings and then an annotated copy of the INF file that shows the problems. Fix all problems that the tool reports. Many of the warning messages point to problems that sometimesprevent the driver from loading properly.

The WDF drivers (UMDF and KMDF) require an additional step to use ChkINF. These drivers require an INX file that the build process converts to an INF. For WDF drivers, build the driver and then run the resulting INF file through ChkINF. Because most of the INX file is present in the INF, you can fix the INX source to remove the problems.

# Debugging and Runtime Checking

As with compile-time checking, taking advantage of the multitude of runtime checks can speed driver development and make your driver more reliable. Runtime checksfindmany problems that appear in drivers. Conversely, because runtime checking affects system performance, these checks can hide some problems that are related to timing in drivers. So be sure to test your driver with runtime checks both enabled and disabled.

You should use all the tools discussed in this section. The errors that the following techniques detect have much less overlap, unlike some compile-time checking techniques that were discussed earlier. Additionally, very few of these tools show false positives. If the problem is reported, it is a valid bug.

Some of these techniques cause the code of the driver to change. We recommend that you follow the following steps in debugging and checking your driver:

1. Activate all checks,such as the checked build and driver verifier, for your driver and work on your driver until it runs cleanly in this environment.

2. Disable the various compile-time modifications and rerun the driver in your debugging environment.

3. Run your driver on the target system (that is, with no debugging enabled) to verify that it works and to use performance profiling against your driver.

## Kernel Debuggers

Microsoft provides two debuggers for working with the Windows kernel:

* KD is a command-line-only debugger.
* WinDbg provides the same command line as KD, but includes a GUI.

The remainder of this section discusses WinDbg. Most of the techniques also apply to KD.

### Setting Up the Debugger

The debuggers can use three possible interconnects: IEEE 1394, serial, or USB 2.0. Each interconnect has special considerations.

#### **IEEE1394**

For debugging over IEEE 1394, you need a standard IEEE 1394 interface on each system and a cable to connect the ports. You cannot use the IEEE 1394 controller for anything but debugging on the test machine. To set up the link:

1. Connect the IEEE 1394 cable to the ports on the two systems.

2. Follow the procedures in “Modifying Boot Settings of the Test Machine”later in this paper to set debug to 1394 and the channel you have chosen for the link.

3. Start WinDbg,click **File**, and then click **Kernel Debug**.

4. On the **IEEE 1394** tab, set the channel to the same value that you used on the test system.

If you encounter problems, disable the IEEE 1394 drivers on the test machine and the IEEE 1394 network adapter driver on the development system, reboot both systems, and retry the debugger.

#### **Serial and Virtual Machines**

Debugging over a serial link requires a null-modem cable to connect your test machine to your development system. The following steps also work for debugging under most VM environments.

For separate machines,the serial port on the test system must be a legacy port becausethe debugger directly manages the port. A nonlegacy serial port that requires a device driver does not work for the system under test. These restrictions do not apply for the development system. On a VM, you must setup a virtual COM port by using a named pipe. To set up the link, perform the following steps:

1. If this is real hardware, connect the null-modem cable to the desired serial ports on the two systems.

2. Test the link with Hyperterminal:

Open a Hyperterminal session to the port with the same baud rate on each system. In a VM environment, on the development system use the named port that you connected to the test system’s virtual COM port.

Use a high baud rate to improve response time for the debugger.

Type in each Hyperterminal session and verify that coherent output appears on the other machine.

Follow the procedures in “Modifying Boot Settings of the Test Machine”later in this paper to set debug to Serial and the baudrate to the fastest setting from the Hyperterminal test.

Start WinDbg on the development machine, click **File**, and then click**Kernel Debug**.

On the **COM** tab, set the baud rate and the port to the values that were used in the Hyperterminal test.

#### **USB**

USB debugging works only on operating systems that are running Windows Vista and later versions and highly depends on your system hardware. Before considering USB debugging, check whether your systems can support it by using the following steps:

1. From the WinDBG directory on the development system, runUSBView.exe and put a well-known device such as a USB flash drive into the first USB port on the system.

2. Check the USBView display to see if the device is in a port of a USB 2.0 Enhanced Host Controller. If so, you might have found the required port.If not, continue through each USB port on the system. If no port is found, the development system is unsuitable. To refresh the USBView display after moving to the next port, press F5.

3. Get USBView.exe from the directory in which you installed WinDBG and copy it to the test machine.

4. Open USBView on the test machine and put a well-known device such as a USB flash drive into the first USB port on the system.

5. Check the USBView display to see if the device is in Port1 of a RootHub of a USB 2.0 Enhanced Host Controller. If so, you might have found the port needed.If not, continue through each USB port on the system.

If this procedure finds a valid port on the development system and a usable Port 1 on the test system, you might be able to debug by using USB. You must now purchase a USB 2.0 debug cable*.* These can be difficult to find and are expensive. To set up the debugging, perform the following steps:

1. On the host machine, plug both ends of the cable into USB 2.0 ports to load the driver that comes with the debug cable.

2. Unplug one end and plug it into Port 1 of the root hub on the test machine that you found with the preceding procedure.

3. To set debug to USB and the targetname to a value that you choose,follow the procedures in “Modifying Boot Settings of the Test Machine”later in this paper.

4. Start WinDbg, click **File**, and then click **Kernel Debug**.

5. On the **USB** tab, set targetname to the name that was used with the test machine.

#### **Modifying Boot Settings of the Test Machine**

The approach for modifying the boot settings varies for Windows Vista and later operating systems and for Windows XP and Server 2003. For all the following examples, the approach is to modify the default boot operating system setting. Other options exist, but this is the simplest. For Windows Vista and later versions, perform the following steps:

1. Issue one of the following commands, depending on the debugging link that you are using, from the command line as administrator:

**Serial:** *bcdedit /dbgsettings serial debugport:1 baudrate:115200*  
This command sets COM1 and a baud rate of 115200.

**1394:***bcdedit /dbgsettings 1394 CHANNEL:42*  
This command sets the channel to 42.

**USB:***bcdedit /debsettings usb targetname:U1*  
This command specifies U1 as the target.

2. Issue the following command:  
 bcdedit /debug ON

3. Reboot the system, and the debugger should connect.

For Windows XP and Windows Server 2003:

1. Issue one of the following commands, depending on the debugging link that you are using, from the command line as administrator:

**Serial:** *bootcfg /debug ON /port COM1 /baud 115200 /ID 1*  
This command sets COM1 and a baud rate of 115200 for the first boot entry.

**1394:***bootcfg /dbg1394 ON /ch 42 /id 1*  
This command sets the channel to 42 for the first boot entry.

2. Reboot the system, and the debugger should connect.

Besides modifying the test machine boot settings, you should consider several other items. For more information, see “Additional Test Machine Setup”later in this paper.

While you modify the boot settings, consider adding boot selections to support varying configurations.Having multiple configurations expands the test coverage for your driver.The following table lists some options to consider.

|  |  |  |  |
| --- | --- | --- | --- |
| Boot.ini | Bcdedit | | Description |
| /3G | increaseuserva | For 32-bit systems, split memory with 3 GB for user and 1 GB for kernel versus the normal even split. | |
| /MAXMEM= | truncatememory | Adjust the amount of memory for the system. | |
| /NUMPROC= | onecpu | Adjust the number of processors for the system. | |
| /PAE /NOPA | pae | Control whether to use physical address extensions (PAEs) | |

#### **Finding Symbols**

The best way to obtain symbols is to use the Microsoft symbol server. This requires a link to the Internet, but it eliminates most of the problems in finding the right symbols. Using the Microsoft symbol server is the only way to obtain the correct symbols for some hotfixes and beta versions of Windows. To set up the symbols, open WinDbg, click **File**, click **Symbol File Path**, and then enter one or more directory paths separated by semicolons, as shown in the following example:

SRV\*e:\SymStore\*http://msdl.microsoft.com/download/symbols;e:\development\mydriver\objchk\_wnet\_x86\i386

This path instructs WinDbg to obtain symbols from the symbol server and cache them in e:\SymStore and, if the symbols are not found, to use the checked build for x86 directory from mydriver.As an alternative, typing the commands *.symfix* followed by *.sympath* in the command window can set up the symbol path.

If you have problems obtaining the symbols, use the*!sym noisy* command to see what is occurring. Set up the source directories in a similar manner,open WinDbg, click **File**, click **Source File Path**, and then enter one or more directory paths separated by semicolons.

### Using Driver Mapping

One very powerful capability of the debugger is mapping driver files. The debugger can intercept the load of your driver on the test system and load the driver from the development system to the test machine. This eliminates the common mistake of forgetting to put the new driver onto the test system.

The simplest way to set this up is to open the Control Panel System application and, on the **Advanced** tab, click the **Environment Variables** button. Under **User Variables**, select **New**, and then enter “\_NT\_KD\_FILES” for the variable name and a full path to a map file that you will create. Next, create the map file that you specified in the environment variable with contents in the format as shown in the following example:

map  
\Systemroot\system32\drivers\mydriver.sys  
e:\development\mydriver\objchk\_wnet\_x86\i386\mydriver.sys

The second line must match the path to the file in the test system's registry. The third line is a pointer to the driver on the development system. Note that using the path to the build directory guarantees that you have the latest driver. You can repeat these three lines for additional drivers.

### WinDBG Display Windows

The WinDbg user interface provides several display windows. This section lists the commonly used windows and provides tipsfor using them. Note that the windows that display data based on the current scope show valid information only when the test system has stopped:

Command window.

This window displays diagnostics and information from commands that you enter. You can search this window by clicking **Edit** and then clicking **Find**, which can help when the window contains lots of data. In addition, you can log the data from the command window to a file by clicking **Edit** and then clicking **Open/Close Log File**.

Sources windows.

These windows display the source code for the driver. First, be sure to set your tab stops to match your editor by clicking **View** and then clicking **Options**. You can quickly set or clear a breakpoint by setting the cursor to the line that you are interested in and then pressing **F9** or the breakpoint toolbar button.

Calls window.

This window shows the call stack on the test machine. Double-clicking an entry in the call stack causes the source window to display the location of the call and the locals windows to show the local variables for the call location.

Locals window.

This window shows local variables for the current scope. You can change the value of a local variable by editing the **Value** field. Variables are displayed based on their types. Setting the **Typecast** button shows the type of each variable. The type in the column can be edited to force the item to be displayed differently.

Watch window.

This window shows variables for the current scope including global variables. Note that, for the global variable to be displayed, it must be accessible from the current scope. The watch window behaves like the locals window.

### Useful Commands

The debuggers have three general types of commands:

Regular commands.

These commands start with a letter and are the core functionality for debugging a driver.

Dot commands.

These commands start with a period and control the operation of the debugger.

Debugger extensions.

These are custom commands that have been created for the debugger. All extension commands start with an exclamation point. Microsoft provides many of these to help you debug drivers. In addition, you can create your own extensions for your driver debugging requirements.

The following list contains some useful debugger commands for debugging drivers. Note that this list does not contain the common commands such as setting a breakpoint or displaying data. Those commands are expected and easy to find. This list provides commands you might not easilyfind:

!error value

This command displays data about the error code that is specified by value.

!analyze-v

This command displays abundant information about a bug check. Whenever the system crashes, you should issue this command to obtain more information about the problem.

.reboot

This command reboots your test system after a crash and the analysis.

lm

This command lists the loaded modules (drivers) in the system. This list can tell you whether your driver was loaded on the test machine.

!devnode 0 1

This command displays all device nodes. If you cannot determine why your driver did not start, use this command.

!irp addr

This command displays the I/O request packet at the location specified by addr.

!poolfind tag

This command displays memory that was allocated with the given tag (see “Pool Tagging” later in this paper).

!wdfkd.help

This command displays various extended commands that are available for KMDF debugging.

.reload [/u] [module]

This command reloads the symbols for the specified module or for all modules if none is specified by name. The /u option unloads the symbols, which is required before you rebuild the driver. If the debugger has the symbols open, the linker cannot write a new file and the build fails.

For more commands, see the debugger documentation.

## Additional Test Machine Setup

While you are setting up the test machine, you should adjustseveral other items to make the debugging experience easier. The following are the major adjustments to make:

Enable debug print output.

Beginning with Windows Vista, debug messages are not displayed by default. To allow debug messages, run the registry editor and find the following key:

HKLM\SYSTEM\CCS\Control\Session Manager\Debug Print Filter

Under this key, create a DWORD value entry that is named DEFAULT and set this value to 0xf.

Enable full memory dumps.

When the system crashes, it is helpful to have the dump available. In the system Control Panel Startup and Recovery application, set the value to **Full Memory dump**. Unfortunately, sometimes the full dump option does not appear. Ifthis happens, find the following registry value and set it to one:

HKLM\SYSTEM\CCS\Control\CrashControl\CrashDumpEnabled

If you want a crash dump on Windows 7 and you have less than 25 GB of free space on disk 1, you need another registry setting. Set the following registry value to 1:

HKLM\SYSTEM\CCS\Control\CrashControl\AlwaysKeepMemoryDump

Disable shutdown event tracker.

On systems that are running Windows Server, a dialog box appears whenever you shutdown the system or the system starts without a clean reboot. This occurs frequently for developers, so it is desirable to disable this capability. Run GpEdit.msc from the windows\system32 directory, and then click **Local Computer Policy>Computer Configuration>Administrative Templates>System** from the left pane of the GUI. In the right pane, right-click **Display Shutdown Event Tracker***,*click **Properties**, and then click **Disable**in the dialog box.

Enable the setup API log.

If you are installing a driver for the first time, you should enable the setup API logging features to be able to locate problems. For details, see “Installation Logging” later in this paper.

Disable User Account Controls (UAC).

Driver development requires many privileged commands on the test system. In Windows Vista and later versions, UAC is continually invoked. There are differing GUI methods for disabling UAC, but setting the following registry value to zero disables UAC on all systems:

HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\System\EnableLUA

Enable keyboard crash control.

When you develop a driver, it is possible to hang the test system. A crash dump helps you understand what is occurring. You can generate a memory dump file by holding down the right CTRL key and pressing the SCROLL LOCK key two times if the following registry key is set to 1:

HKLM\SYSTEM\CurrentControlSet\Control\CrashControl

Set the symbol path.

If you want to use the *kernrate* or *XPerf* performance tools that are described later in this paper, set up a system environment variable with the path for symbols for the system and your drivers. To do this, set the **\_NT\_SYMBOL\_PATH**environment variable to a string such as the following:

symsrv\*symsrv.dll\**cache*\*http://msdl.microsoft.com/download/symbols;*path*

where *cache*is a directory path to a directory in which to cache the symbol files and *path* is the directory in which you will put your driver’s symbol files. Also be sure that *symsrv.dll* is on the test machine.If not,obtain it from the Windbg directory on your development system and put it in the *windows\system32* directory.

Add *wttlog.dll* to the system.

Several test tools in the WDK rely on *wttlog.dll*. If you want to use the testing tools, copy *wttlog.dll* from *\bin\<arch>*of the WDK to the test system’s *windows\system32* directory or to a similar location on the execution path.

Fortunately, two free tools let you adjust the debugger and most of the preceding settings in one tool. The tools, OSRSysTool and BellaVista, are listed in the “Resources” section.

## Checked Build of Windows

The checked build of Windows is one of the most powerful diagnostic tools that areavailable to a driver developer. Some individuals confuse the checked build of their driver with the checked build of Windows. The checked build of Windows is a version of the operating system that was built with checking turned on, in the same way that you can build your driver with the checked build. You should not ship a driver and support code unless the product runs cleanly on the checked build of Windows. The following are several useful features of the checked build:

* Extensive assertions, particularly to check parameters to functions and the internal state of Windows.
* Significant diagnostic output that can be enabled for specific features.
* Most optimizations disabled. This makes it easier to debug into a call to the kernel and changes the timing of routines, which can help find timing-dependent problems in your driver.

### Assertions

The checked build uses the same ASSERT macro that driver writers use. The macro reports into the debugger the expression that failed and the file and line number at which the failed expression appears in the source code. The following is an example report:

\*\*\* Assertion failed: Irp->IoStatus.Status != 0xffffffff  
\*\*\*   Source File: D:\nt\private\ntos\io\iosubs.c, line 3305  
  
0:Break, Ignore, Terminate Process or Terminate Thread (bipt)? **b**  
0:Execute '!cxr BD94B918' to dump context  
Break instruction exception - code 80000003 (first chance)  
ntkrnlmp!DbgBreakPoint:  
804a3ce4 cc               int     3

In this example, the user entered a “**b**” to stop at a breakpoint.Setting a breakpoint allows you to check the stack and see where your driver is calling into the kernel, which is the likely location of the problem.

Thousands of ASSERT statements in Windows are active in the checked build. In general, they fall into three major categories:

Parameter checks.

These can be simple range checks for a value that is passed to a kernel routine. Another parameter check uses the type data that many kernel data structures contain to verify that a pointer to the correct type of data is passed.

Timing actions that have recommended limits.

These useful ASSERT statements check the timing of actions for which Microsoft has recommended limits. For instance, holding a spin lock for longerthan the recommended time triggers an assertion failure.

Checks on internal data structures.

These checks verify internal Windows data structures. These failures are difficult to find because the assertion is rarely triggered until long after the actual error.

Unfortunately, the condition of the ASSERT did not appear in Windows Vista and Windows Server 2008. It does appear in Windows 7 and later versions. Even without the detailed information, ASSERTs are still valuable. If you cannot determine the problem from looking at how your driver calls code that leads to the ASSERT, consider asking about the specific ASSERT (based on the location) on a Microsoft driver forum. Typically, you will receive data on what is being triggered.

### Diagnostic Output

The checked build of Windows provides support for diagnostic output to the debugger. If you set the default value that was documented in the “Additional Test Machine Setup” section earlier in this paper for enabling debug print output, all components report diagnostic information. The type of information varies depending on the component and the version of Windows that you are running.The Knowledge Base article "How to: Enable Verbose Debug Tracing in Various Drivers and Subsystems" shows how to get data that helps diagnose many problems for a few of the possible categories of output. Look up DPFLTR\_TYPE in dpfilter.h for all categories.

Most of this output must be enabled. For example, the following debugger commands enable kernel and user mode Plug and Play manager diagnostics:

ed NT!Kd\_NTOSPNP\_Mask 0xFFFFFFFF

ed NT!Kd\_PNPMGR\_Mask 0xFFFFFFFF

Enabling these diagnostics can help locate many driver problems.

### Installation of the Checked Build

You can install the checked build of Windows in either of two ways:

* Install parts of the build on a standard Windows system.
* Install the complete checked build in its own disk partition.

Each installation has challenges and advantages.

The partial installation is documented in *“*Installing Just the Checked Operating System and HAL” in the WDK*.* This technique installs just the checked versions of the kernel and the hardware abstraction layer (HAL) with the free versions of other system components. This has the advantage of running much faster than the full checked build, but limits checks and diagnostics. In particular, many of the subsystem outputs that are described in “How to: Enable Verbose Debug Tracing in Various Drivers and Subsystems" require the drivers from the checked build. Additionally, many of the drivers that are supplied with the checked build have assertions that can help you find problems in your driver.

The full installation requires its own disk partition. The full checked build of Windows is installed like any other version of the operating system. The full checked build runs significantly slower, but provides many more assertions and additional diagnostic output. One problem with the full checked build is that it might assert during installation.If you encounter an assertion, you must install the checked build with the debugger running, so that you can ignore the assertion and continue. If you do encounter an assertion, you must always boot the checked build while the debugger is running. How to perform the install with the debugger running depends on the operating systemthat you choose:

Windows Vista and later versions.

You should install these versions from a running system. If necessary, install the retail build of the operating systemthat you want, and then replace it with the checked build. Place the installation DVD in the drive and change the directory to the root of the DVD. Then issue one of the following commands:

setup.exe /1394debug:1

setup.exe /debug:1 /baudrate:115200

setup.exe /usbdebug:testmachine01

The first command sets up debugging over 1394, the second command sets up debugging over COM1 at baudrate 115200, and the third command sets up debugging over USB. The debugger connects when the installation first reboots.

Windows Server 2003.

If your test machine has two serial ports, you can start debugging directly from the installation disk. Otherwise, follow the instructions for Windows XP. When you boot the media to install the system, you can press the F8 key when installation first starts. When you are prompted for F6, press the F8 key. This activates the debugger over the COM2 port with a baud rate of 19200.

Windows XP.

You must install the system from a running version of Windows. Invoke the installation program with the /noreboot option. After the GUI installation program finishes, find the temporary directory where the files were installed (usually C:\) and edit the txtsetup.sif file. In this file, find the following line:

OsLoadOptions = "/fastdetect /noguiboot /nodebug"

Edit this line by removing the /nodebug option and adding the debugger options that you would use in the boot.ini file. When you reboot to continue the installation, the message “Starting Windows” appears in the white line at the bottom of the screen and the system starts in the debugger.

## Driver Verifier

Driver Verifier, like the checked build of Windows, is a powerful tool for finding driver problems. Whereas the checked build is passive (only adding checks to the normal operation of the kernel), Driver Verifier is an active checking tool that can modify the environment to test a driver. Additionally, Driver Verifier provides monitoring capabilities for the drivers under its control. Driver Verifier ships as part of the Windows operating system and can check the drivers that you choose.

Driver Verifier is controlled by the Verifier utility. This utility has too many options to explain in detail in this paper. All options are documented in the WDK under “Driver Verifier Manager (verifier.exe).”For most drivers, you should always enable the following options:

Automatic checks.

These standard checks always apply to any driver that is being verified and cannot be disabled.

Special memory pool.

This option allocates memory from a special pool to check for buffer overruns and related problems.

Forcing IRQL checking.

This option checks memory references and finds attempts to reference memory at the wrong IRQL.

Memory pool tracking.

This option tracks your driver's memory usage and ensures that the driver frees all memory.

I/O verification.

This option monitors your driver's handling of I/O requests.

DMA verification.

This option monitors your driver's handling of DMA operations.

Deadlock detection.

This option finds many potential deadlocks by ensuring that your driver uses a consistent locking model.

Other options are available for specific types of drivers. The important thing is to enable Driver Verifier for your driver before you ever load your driver. This way, you begin to find bugs early in the development process.

Driver Verifier also maintains several counters that you can monitor. These counters provide information about memory allocation and spin locks. Both the Verifier utility and the!verifier debugger extension can display these counters.

Driver Verifier has one option that you should not enable when you first test your driver. This option is low resource simulation,which injects failures into your driver’s requests for resources. Testing with this option is important because your driver should not crash the system if it cannot allocate a resource such as memory. However, you should use this option for testing after your driver is running well by using the techniques in this paper. Note that,when you use the command line, you can obtain finer control of this option and the verifier in general. To obtain all options, use verifier /?.

## WDF Tools

WDF was designed to reflect the current best practices of the industry. As part of the capabilities,it provides several tools to aid debugging these drivers. These tools are another reason that you should develop your driver by using WDF if possible. This discussion assumes that you are using a recent version of WDF with the latest tools.

### WDF Driver Verifier

WDF Driver Verifier provides checks that are tuned to KMDF and UMDF drivers beyond those of Windows Driver Verifier. Good practice is to enable both Windows Driver Verifier and WDF Driver Verifier for your driver from the first time that you load it. In WDF 1.9 and later versions, the WDF Driver Verifier is automatically enabled if the Windows Driver Verifier is enabled for the WDF driver that is being verified.

The easiest way to activate the WDF Driver Verifier is to use *WdfVerifier.exe,* whichcomes with the WDK. Copy this file from the *tools\WDF\<machine arch>* directory of the WDK to your test machine. Invoking the program displaysseveral options.The exact form of these options can vary depending on the WDF version. Some of these options are for controlling logging, which is covered in the next section. As with Windows Driver Verifier, you should enable all options except memory allocation failure. As with low resource simulation, you should enable memory allocation failureonly after the driver works correctly for normal operation.

### KMDF Logging

KMDF logging acts like an in-flight recorder for your driver. The system shows you what occurred in your driver for the last several operations of the framework. The simplest way to set up logging is to use *WdfVerifier.exe* and enable logging.You might want to consider verbose logging to obtain all possible data.Be aware that such logging can fill up the log quickly. To view the logging, you should have WinDBG running and then use the following commands:

!wdftmffile*TMFfile*

!wdflogdump*driverName*

The first command sets the path to the trace messages. This is a file that is named *wdfXXXXX.TMF* in the */tools/tracing/<machine arch>*directory of the WDK. The *XXXXX* in the path name is of the following form: the first two digits identify the major version of WDF, followed by three digits for the minor version. Choose the TMF file for the version of WDF that you are using on the test machine. Be sure that this version matches. You use the second command whenever you want to see the log for the driver. Be sure to use the name of the driver without the *.sys* extension.

You can monitor the tracing on the test machine in two different ways. The simplest way is to use *wdftrace.cmd*. This command can provide information when WinDBG is not available. Alternatively, you can look in *wdftrace.cmd* for the provider globally unique identifier (GUID) andthen use that GUID and the TMF file with the various WPP tracing tools that are discussed later in this paper. Typically, we recommend that you use Windbg to obtain the log.

Finally, you can modify the prefix of KMDF logging. In Windbg, use *!wdfsettraceprefix* to set the string that you want. Or you can use the TRACE\_FORMAT\_PREFIX environment variable to set it. In any case, look at the WinDBG documentation on *!wmitrace.setprefix* for the options that you can use.

### WDF Fault Injection

WDF provides some limited fault injection that is called WDFTester*.* This tool is in the WDK’s *tools\WDK\WDFTester\<machine>*directory. WDFTester provides two major capabilities: the ability to log all calls in KMDF that return a status value and the ability to insert a failure into a given type of WDF device driver interface call.

To use either of these calls, copy the WDFTester directory for the test system’s architecture to the test machine and put a copy of devcon from *tools\devcon\<machine>*in the same folder. The operations of WDF tester are controllable by WDFTesterScript, whichyou canrun as shown in the following example:

cscript WdfTesterScript <arguments>

The following table shows the arguments of the script.

|  |  |
| --- | --- |
| install | Install and start WdfTester. |
| register <drivername> | Register a driver for test. |
| unregister <drivername> | Unregister a previously registered driver. |
| getddistats<drivername> | Return a list of WDF calls that can return status and their count. |
| WaitForFiEvent | Wait for a fault injection event to occur. |
| enableddifi<drivername><ddiname> | Enable a specific WDF call to be fault injected. |
| disableddifi <drivername><ddiname> | Disable a previously enabled call. |
| runtest <drivername><hwid> | Execute fault injection test. |

After registering a driver, we recommend that you disable the driverand then enable it to be sure to view all calls. If you have all KMDF output on from the verifier, then after a driver is registered,every call to the framework that can return an error is logged in the debugger. The amount of this output can be overwhelming.

The simplest way to use the tool is to install the tool and then invoke the *runtest* option.

## Pool Tagging

Pool tagging allows your driver to associate a four-character alphanumeric tag with each block of memory that you allocate. The usual approach to use pool tagging is to create a unique tag for each class of data structure that you allocate. Be aware that none of the WDK tools checks to ensure that the tag you create is unique for the system. You can see the tags that Windows is aware of by viewing PoolTag.txt, which is under the WinDbginstallationdirectory in the *triage* directory. Try to make your tags distinct from common tags such as “*Irp”*or “*Ddk”.*

The actual tagging of memory is enabled according to some unusual rules. Tagging is never enabled if the Driver Verifier “special pool” option is being used. Windows 2000 and Windows XP enable tagging by default only for the checked build of Windows. For the free build, you can enable tagging by using the following command line:

gflags /r +ptg

Tagging is always enabled for all builds of Windows Server 2003.

These tags and their allocations can be viewed in three ways:

* In the debugger.
* By using the Poolmon utility that is supplied with the system.
* By using the Pooltag utility that is supplied with the WDK.

Pool tagging letsyou track the memory allocations of your driver for specific data structures. Tracking is especially valuable when Driver Verifier reports that you have not freed all the memory that you allocated when your driver unloads. Looking at the tags from the memory that is being reported makes it easier to determine which allocations you forgot to free.

Beginning with Windows XP, in addition to allocating memory with unique tags, you can also supply that tag when you free memory by using ExFreePoolWithTag. If you free memory by using ExFreePoolWithTag, the tag that the call supplied is checked against the tag of the memory that you are freeing. Mismatched tags cause the system to bugcheck.

Allocation and freeing with tags provides type checking for memory. Verifying that the memory to be freed is of the intended type can finderrors that were created when you freed the wrong memory. Bugs from incorrect freeing of memory can be some of the most difficult to find.

Typically,the memory can still be freed without specifying the tag. Drivers can require the use of ExFreePoolWithTag by ORing the tag with the PROTECTED\_POOL flag. If your driver is for Windows XP and later versions, you should use ExFreePoolWithTag with PROTECTED\_POOLtags except when you allocate memory that the system will free. In those cases, do not use a PROTECTED\_POOLtag.

## Code Coverage

Knowing what parts of your driver have been executed is invaluable to achieving a quality device driver. Although many developers rely on ad-hoc methods such as a trace statement, there are tools to obtain information on code coverage.

### DrvCov

With the Windows 7 WDK, Microsoft released a tool that shows you what I/O requests are going to or coming from a driver. DrvCov is located in the *tools\drvcov\<machine>*directory of the WDK. Copy the correct architecture version to your test system, and then run the tool from a command prompt as the administrator. This tool uses a filter driver that must be installed, followed by a system reboot, to collect data. The following table lists a simplified command list for DrvCov.For complete information, see “Driver Coverage Toolkit” in the WDK documentation.

|  |  |
| --- | --- |
| drvcov /da | Lists the devices on the system. |
| drvcov /a <id> | Installs the filter to monitor input requests to the device. Id is either the device node from the precedingcommand or a device Id in quotes (“”). |
| drvcov /al <id> | Installs the filter to monitor output requests from the device. |
| drvcov /c <id> | Displays the coverage on the console. |
| drvcov /e <id> | Zeros the coverage counters for the device. |
| drvcov /r <id> | Removes the filter for the device. |

The common usage is to list the devices and then choose one or more to monitor. Reboot the system, exercise the chosen device, and then display the coverage.

### Statement Coverage

Commercial tools are available to provide coverage data to indicate if a given statement, or even a portion of a condition statement, is executed. The more of your code that is executed during testing, the more likely you are to find errors.

The best way to check statement coverage is to run all your tests for a driver with code coverage enabled and then see how much of your code was exercised. Do not be surprised if you run statement coverage with all your tests and find that roughly two-thirds of your driver code is not being executed. Increasing statement coverage involves the following approaches:

Checkthat your tests make all valid requests.

With statement coverage, you see what code is actually run. It is easy to see valid requests to your driver that are never tested. Most test suites need improvements to increase coverage.

Check that your tests supply all incorrect input.

Just as with valid requests, you are likely to find that your test code does not try all possible incorrect requests for your driver. When you review this code, be sure to check whether your driver handles all incorrect requests correctly.

Use low resource simulation in Driver Verifier.

Another way to increase coverage is to use the low resource simulation option in Driver Verifier to exercise failure paths in the driver.

Use WDF fault injection.

If your driver is WDF, use the fault injection tool to help increase the coverage area.

Consider adding fault injection to your driver.

Even after this work, you are likely to find a lot of code that is not exercised. Typical examples are failures in your hardware that are difficult to create or similar “impossible conditions.” For these, you might need to create a macro to inject faults.

A simple fault injection macrocalls a function that randomly returns TRUEfor the debug version of your driver. In the production version of your driver, the macro always returns FALSE. You then add code to your driver that, when the macro is true, modifies the behavior of the driver to execute the failure condition, as shown in the following example:

devStatus = READ\_REGISTER\_ULONG( statusRegister );

devStatus |= FaultInject ? DEV\_ERROR\_FLAG : 0;

if ( devStatus & DEV\_ERROR\_FLAG ) …

Adding the second line makes it possible to simulate the situation in which the device returns the error flag and thus exercises the recovery code that the third line controls.

Even after you do all this, you still will not obtain complete coverage. The goal of statement coverage should be for you to see that all reasonable cases are being tested. You should thoroughly check the code that is not exercised to find problems.

## Performance Profiling

KernRate is an often-overlooked tool in the WDK. It isprimarily a profiler that periodically samples the instruction pointer. KernRate also collects statistics on several performance-related items in the kernel such as those in the performance monitor (Sysmon).Although KernRate has been supplemented by Xperf, KernRate still offers some capabilities that make it a desirable tool for developers to know.

If you have not done performance profiling before, remember that obtaining good data takes time. For good solid numbers, consider running a profile for hours. In addition, your numbers are worthless unless you are exercising your driver, so you need a good test that can exercise it for a long time period. Besides time, the profile takes a lot of space.Sample profilers look at the instruction pointer and place the data into a “bucket” that indicates the code that is being executed. KernRate allows you to adjust the size of the buckets.The smaller the bucket, the more accurate the sample.

One difficult characteristic of KernRate is the large number of options that are available with the command. To get started with profiling, try the following command line:

kernrate -b 4 -r -x -v 1 –z <driver> -s <seconds>

This command provides the required typical data. You should replace <driver> with the name of your driver file and<seconds> with the number of seconds that you want to run the profile. Before you run this command, be sure that the symbol path is set up on the system as described in “Additional Test Machine Setup”earlier in this paper.

Run a performance profile of the entiresystem while you exercise your driver. Even with a significant load on your driver, the percentage of time spent in the driver should be relatively low for most situations. When you analyze the data from this profile, look for hot spots where your driver takes anextended time period. At each hot spot, look for either a problem with the code or a better algorithm that reduces the load. Many hot spots are valid, but checking them can find bugs.

## Kernel Logging

As part of Event Tracing for Windows (ETW), recent kernels can log severalsystem actions. We will discuss considerations for ETW in your driver later in this paper, but for now let us look at the data that the system logged. The simplest way to log this data is with the TraceView utility that is provided in the WDK tools directory. TraceView can log nine different categories of data, including:

* Process creation and termination
* Thread creation and termination
* File I/O
* Disk I/O
* Image file (executables and DLL) load and unload
* Registry accesses

Enabling kernel logging while debugging your driver can help find bugs.

## Windows Performance Analyzer

Windows Performance Analyzer, also known as XPerf, is a new tool that can provide data on kernel performance. XPerf uses ETW, which expands both the collected data and the ways to display it.

Xperf is part of the Windows Software Development Kit (SDK). You must get the latest SDK and install the Win32 tools. After you install it under the SDK *bin* directory, find the installation files that are named *wpt\_x86.msi, wpt\_x64.msi,*and *wpt\_ia64.msi*.Install the filethat is appropriate for your test system.

Xperf is a complex tool about which several papers have been written. For kernel values you must run Xperf with the Administrator account. To get started with the Xperf for kernel data, issue the following command:

xperf –providers K

The preceding command displays all available kernel information under *Kernel Flags* and lists predefined groups of flags under *Kernel Groups.* Use a collection of kernel flags separated by plus signs (+) to start a trace, as shown in the following example:

xperf –on FileIo+CSWITCH

The preceding command starts a trace with the FileIo group plus CSWITCH (context switch). When you want to stop the kernel trace, use the following command:

xperf –stop

The default for the preceding commands is to write a log file in the system root that is named *kernel.etl.* To view the data from the trace, use the *xperfview* command.This tool allows you to graphically display the collected data.

A powerful feature of Xperf is the ability to capture and display stack walks on events. To obtain a list of events that can have their stack captured, issue the following command:

xperf –help stackwalk

The preceding command displays a list of all events that can collect stack walks*.* The command to start a trace can include stack walk information as shown in the following example:

xperf –on FileIo –stackwalk FileRead+FileWrite

You can stop the trace in the same way as before and view it by using *xperfview.* If you right-click the graph and look at the summary table, you see the modules that initiated the actions.

As stated earlier, Xperf is a complex tool with many capabilities. For more information on how to obtain the tool and how to use it, see the MSDN website.

## Installation Logging

Just as you used ChkINF to verify your INF file before installation, you should enable SetupAPI logging during driver installation. Driver installations perform several actions. Logging provides a way to validate that your installation is correct. If your driver installation fails, the log is available to show you what occurred. To enable Setup logging, you must set the following registry DWORD value to 0x00007070for Windows Vista and later versions, and to 0x3000FFFF for earlier operating systems:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurentVersion\Setup\LogLevel

This setting enables verbose logging so that you see everything that occurs during device installation. Be sure to restore the value at the end of the installation because verbose logging produces a lot of output, which slows down the system.

The setup log resides in %*systemroot*%\inf\*setupapi.dev.log* for Windows Vista and later versions and in %*systemroot*%\*setupapi.log*for earlier versions*.* Examinethis file to see what your device installation did. This log contains a lot of data. An excellent reference for understanding it is “Debugging Device Installation in Windows Vista” for Windows Vista and later versions and "Troubleshooting Device Installation with the SetupAPI Log File" for earlier versions. Both documents can be found on the WHDC website.

## Miscellaneous Tools

Windows and the WDK include several other miscellaneous tools that many developers overlook:

DriverQuery

DriverQuery is a command-line tool that ships with Windows. This tool lists all drivers that are active on the system. Using the /v option with this tool expands the data to include the size of the driver, the path to the driver, and other information. Applying the /si option provides data about whether the driver is signed and what INF file installed the driver on the system. Use DriverQuery to see that your driver is loaded and to verify that the INF file is what you think it is.

DeviceTree

DeviceTree is a GUI-based tool that displays data about drivers and devices. In the Driver view, the tool lists all drivers, giving data on the driver, the calls that it supports, and all devices that it created. Under the driver, DeviceTree displays data about each device, including the flags on the device object and its reference count. In the Plug and Play view, the device data appears in a tree that shows the relationships among devices. DeviceTree can help identify several problems when you debug your driver. This tool no longer ships with the WDK, but you can download it from OSR Online.

DevCon

DevCon is a command-line tool that provides all capabilities of Windows Device Manager. In addition, it provides many of the capabilities of the DriverQuery and DeviceTree tools. DevCon is useful for creating scripts to exercise Plug and Play control of your device. The source code for DevCon is supplied in the WDK in *src\setup\devcon*. Use this code as a guide to writing your own driver tools.

RegEdt32

Installing drivers modifies the registry. Many drivers use data from the registry to control operation. RegEdt32 is the tool to use to inspect and edit the registry, including the access control for registry keys.

In addition to these tools, many third-party tools are also available to driver writers. For more information about third-party tools, see “Resources.”

# Diagnostics

Diagnostics is often overlooked but is critical to a high-quality Windows device driver. It is the only area of coding that is covered in this paper.

It is important to have a standard diagnostics model for all drivers that your company produces. The model should include common levels and flags to control diagnostics. The model should also have a standard presentation format for diagnostics. This presentation format should consider everything from standard naming for event log entries and performance monitoring counters to displaying data for assertions, debug prints, and WPP tracing.

## Assertions

Earlier in this paper, we encountered assertions with the checked build of Windows. The use of assertions is a powerful debugging technique, andyou should be aggressive in using them. The assert statements both provide debugging capabilities and document the assumptions that your code makes.

The WDK provides several macros so that your driver can assert when it is built in the checked build environment:

ASSERT ( expression )

ASSERTMSG ( message, expression )

NT\_ASSERT ( expression )

NT\_ASSERTMSG ( message, expression )  
NT\_ASSERTMSGW ( message, expression )

In each macro, the expression is checked and, if it is false, the assertion is fired in the debugger. The ASSERTMSG variants allow an additional message to be printed as part of the diagnostics, with the string being ANSI text except for NT\_ASSEERTMSGW, which is UNICODE. These macros return the value of the expression and produce code only in a checked build of your driver. They have no impact on your production code.

The NT\_ASSERT variants use the new mechanism that requires the debugger symbol file to see the information about the ASSERT. Using the NT\_ASSERT variants makes drivers smaller and lets you keep the message from the customer if you want to.

Additionally, the following are two other assert macros in the WDK:

RTL\_SOFT\_ASSERT ( expression )

RTL\_SOFT\_ASSERTMSG ( message, expression )

Unlike the other ASSERT macros, these just test the expression and print the data but do not break into the debugger.

Finally, the following are assertions for use only in KMDF drivers:

WDFVERIFY( expression )

VERIFY\_IS\_IRQL\_PASSIVE\_LEVEL ()

These assertions trigger only when WDF Driver Verifier is enabled.

You should use assertions whenever you enter or exit a major block of code. The following are several examples:

On entry to and exit from a module of code with static data.

Although many developers do not think of modules in C programming, a file that has static data is a module. On entering any non-static function, the module should validate its static data with asserts. On exit from a non-static function, the module should perform assertions on the static data to verify that the module did not corrupt the data.

On entry to and exit from a procedure.

The common use of assertions is to check the validity of input parameters to a function. What many developers forget is that you must validate the data that you touch on the exit of your function.

On entry to and exit from a loop.

Just as you perform checks around procedures, it might be appropriate to perform checksaround a loop. Checking pre-loop and post-loop conditions can help to localize problems.

Before data is referenced by a pointer or before you use a global variable.

Many developers check a pointer before referencing it, but you should also validate the data to which the pointer points. In the same manner, you should also validate the contents of global variables before you use them.

Given the places that you need to assert, your checks should entail the following common validations:

Pointers are valid.

A common assertion check is that a pointer is not NULL. This check is a good start, but further checks are necessary. For instance, many Windows kernel structures have a type field. In this case, you should also validate the type. For example:

ASSERTMSG ( “Invalid IRP pointer”,

irp != NULL && irp->Type == IO\_TYPE\_IRP)

You can go further and check whether the memory that is pointed to is writeable, for instance. Do not stop with the simple check for NULL.

A variable is in range.

Another common check is that a given value is in range. The goal is to be sure to validate all variables that you obtainfrom outside your code.

Doubly linked list is consistent.

Before you think this is excessive, consider the following assertion:

ASSERTMSG ( “Corrupted List”,

listPtr != NULL && listPtr->Flink != NULL &&

listPtr->Flink->Blink == ListPtr )

The additional checks can findlist corruption. Adding the assertions does not find errors as quickly as validating every element of the list, but it is simpler and haslittle overhead.

String variables are valid.

Many Windows strings are counted strings, in which the string is designated by a structure with a buffer pointer, length, and maximum length. When you validate a string variable, consider the following assertion:

ASSERTMSG ( “Invalid Counted String”, string != NULL &&

string->MaxLength > string->Length &&

string != NULL ?

(string->Length != 0 ) : TRUE ))

The additional checks make sure that you do not encounter an invalid string.

IRQL is valid for a function.

This check is driver-specific and is an important one. Many kernel service routines work at a limited range of IRQL. Therefore, you should ensure that the functions that call these routines check for the correct IRQL.

Developers make the following two common mistakes when using assertions:

Introduce side-effects that modify driver behavior.

Be sure that your assertions and any other checking code do not produce side-effects that modify the behavior of your driver. It is easy to mistakenly write code such as the following example:

ASSERT ( --CurrentCount > 0 )

This assertion workscorrectlyuntil you move to the free build and find that your counter is not being decremented.

Confuse assertions with error checks.

Donot use an assertion for a condition that might occur during normal operation, such as the following example:

p = ExAllocatePoolWithTag ( NonPagedPool, BLK\_SIZE,   
 BLK\_TAG );   
ASSERT ( p != NULL )

This usage is incorrect because in normal operation your allocation might fail for lack of memory.

You can make many more validation checks in your code. Remember that you can make these validations complex because these assertions exist only in the checked build of your driver. In fact, consider creating functions that validate complex data structures and place these functions in conditional code so that they exist only in the checked build of your driver. This technique can help you find bugs early.

## Debug Print Statements

Debug print statements are the most common diagnostic in most drivers. The WDK describes four debug print statements:

* KdPrint (format, ... );
* KdPrintEx (componentId,level,format, ... );
* DbgPrint (format,... );
* DbgPrintEx (componentId,level,format, ... );

All these calls use the same *format* and arguments as the printf statement. The first two calls are macros that produce code only in the checked build of your driver. The last two calls are the underlying calls that are always active. For the Ex versions of these calls, use componentId and level to allow filtering the debug output.

Your driver should not use the Dbg versions of these calls unless you are creating your own conditional macro for printing. Using debug print statements incurs significant overhead, and they should not appear in a production driver.

Even in the checked version of your driver, having all debug prints active can make your driver extremely slow to use, so consider creating your own macro to allow conditional debug prints. This is the reason that Windows Vista and later versions disable the debug prints by default as was discussed in “Additional Test Machine Setup”earlier in this paper.

Before you decide to use only DbgPrintEx, you should be aware that a limited number of component IDs are available for driver developers, so it is better to create your own macro, such as the following example:

#if DBG

#define MyDriverPrint ( level, flag, args ) \

if ( MyDriverLeven >= level && \

(flag) & MyDriverFlags ) \

{ \

DbgPrint args; \

}

#else

#define MyDriverPrint ( flag, level, args )

#endif

This example creates a macro that prints only for the checked build of the driver. The debug flags and level are global variables that your driver can initialize to control its output. Using this macro matches the model of WPP tracing that is discussed in “WPP Software Tracing” later in this paper along with the advantages of keeping a constant model.

Typically, debug print data is monitored in WinDbg. DebugMon and DebugVieware two tools that are available to monitor debug messages on the system with your driver. For availability of these tools, see “Resources.”Both tools provide a GUI to monitor debug messages on a local system or over the network. Do not assume that these tools and debug prints are enough to debug a driver. As discussed in this paper, many verification tools are available and you need the debugger to take full advantage of these tools.

Having looked at the debug print calls, consider the following simple rules for your output:

Use text whenever possible.

For example, it is much easier to read “IRP\_MJ\_CLOSE” than to see 0x2 and remember that 0x2 indicates a close request. Several drivers in the WDK samples have example code for printing IRP major functions.

Print information that describes a data structure, not just a pointer to the structure.

For example, print the major function of the IRP and the pointer to the IRP. Having the function code makes it easier to read and act on the data.All too often, debug statements vary widely, which makesthe output difficult to use while you are debugging your driver.

Consider providing a call trace that shows the entry to and exit from functions in your driver.

A good approach is to use an indent level so that nested calls appear indented from their callers.

Consider adding more data to your debug print macro.

Having a standard summary line that shows the source file and line, the currently executing thread, and the IRQL can help debugging.

As previously noted, the format string follows the conventions of printf. Consider the following tricks and rules for format strings:

* If you are printing a pointer, use %p to correctly print both 32-bit and 64-bit variables.
* If you are printing a counted string, use %Z (ANSI) or %wZ (Unicode) to print the string correctly even if it is not null-terminated.
* If you are printing an integer whose size can be either 32 or 64bit, depending on the build environment, use %I to specify the size of the variable. Likewise, if you have a 64-bit integer, %I64 specifies this size.
* If you are trying to print a Unicode value, remember that the driver must be running at an IRQL that is less than DISPATCH\_LEVEL.
* When writing your debug statements, realize that all these functions have a limit of 512 bytes per call.

## WPP Software Tracing

WPP software tracing provides a low-overhead mechanism for logging data. It creates a binary data log that is post-processed into a set of text messages. A significant advantage of using WPP tracing is that your customers cannot read the data that is collected from your driver. This capability lets you use messages that might otherwise be embarrassing such as “I guess the hardware can return this code after all.”

The simplest approach to using WPP tracing is to add the following line to the end of the SOURCES file:

RUN\_WPP=$(SOURCES) -km

You must then set up some definitions and Include files in your driver. Finally, add statements of the following form:

DoTraceMessage(IN TraceFlag, IN Format, ...)

Format in this statement is the same as that for debug prints. DoTraceMessage is the default name of the trace statement. The WPP tracing tools let you define your own name so, for example, you can convert debug print statements into trace statements. It is also possible to force WPP tracing output as debug print statements. To make WPP tracing also appear as debug print statements, use the following command:

#define WPP\_DEBUG(*args*) DbgPrint args;

This statement still requires you to enable WPP tracing to see the messages.To always enable the trace, you can add the following code to your driver immediately after the call to WPP\_INIT\_TRACING:

#if DBG

WPP\_CONTROL(WPP\_BIT\_FunctionTrace).Level = TRACE\_LEVEL\_VERBOSE;

WPP\_CONTROL(WPP\_BIT\_FunctionTrace).Flags[

WPP\_FLAG\_NO(WPP\_BIT\_FunctionTrace)] = 0xFFFFFFFF;

#endif

You might not want to use WPP\_DEBUG, however, because WPP tracing supports several format strings that are not supported in debug prints. The following table shows some of the more interesting format strings.

|  |  |
| --- | --- |
| %!FILE! | Displays the name of the source file. |
| %!FLAGS! | Displays the value of the trace flags. |
| %!LEVEL! | Displays the name of the trace level. |
| %!LINE! | Displays the line number of the line. |
| %!bool! | Displays TRUE or FALSE. |
| %!irql! | Displays the name of the current IRQL. |
| %!sid! | Takes a pointer to a security identifier (SID) and displays the SID. |
| %!SRB! | Takes a pointer to SCSI request block (SRB) and displays the SRB. |
| %!SENSEDATA! | Takes a pointer to SCSI SENSE\_DATA and displays the data. |
| %!GUID! | Take a pointer to a GUID and displays the GUID. |
| %!NTSTATUS! | Takes a status value and displays the string that is associated with a given status code. |

Additional formats plus the ability to display enumerations are described in “Software Tracing FAQ”in the WDK.

WPP tracing presents some challenges to the developer in designing output. In particular, WPP tracing achieves lower overhead by keeping data binary. Some guidelines for debug print statements do not apply to WPP tracing. The recommendation to convert constants to text is an example. Placing a binary 0x2 in the message takes one instruction, whereas “IRP\_MJ\_CLOSE” requires a string copy. Also, including the summary data is unnecessary because WPP tracing already includes the file and line data.

WPP tracing provides diagnostic support for its actions. To enable this support, add the–v4 option to the RUN\_WPP line in the SOURCES file. To display this data, define WppDebug to be a debug print statement, as shown in the following example:

#define WppDebug(\_level\_,\_msg\_) KdPrint (\_msg\_)

This statement causes the state transitions of WPP tracing to appear as debug prints. This debug output lets you monitor the operation of WPP tracing and see that your requests to enable and disable tracing are correct.

Now that we have discussed WPP tracing, you should be aware of several tools that are available for controlling and displaying tracing. The following table lists those tools.

|  |  |
| --- | --- |
| Logmon | This tool controls tracing (such as stopping and starting). The tool ships with Windows XP and later versions. |
| Tracelog | This tool controls tracing. The tool ships with the WDK, SDK, and symbols. The source for the tool is available in the SDK, so you can write a custom tool if you want to. |
| TraceFmt | This tool translates a trace log to a readable format. The tool ships with the WDK, SDK, and symbols. |
| TracePdb | This tool extracts the trace format data from a symbol file. The tool ships with the WDK, SDK, and symbols. |
| Traceview | This is a general tool for controlling and displaying trace data. The tool is shipped in the tools directory of the WDK. |
| WmiTrace extensions | The debuggers have an extension to provide control and display of trace data. |

WPP tracing is a powerful tool that you should use in every shipping device driver. The low performance effectof tracing provides you with a way to obtain detailed diagnostics from a customer without requiring a modified driver.

## Event Log

So far, this paper discussed diagnostic data that is designed for the developer. It is just as important for your users to know what is occurring. The best way to provide this information is through the System Event Log.

An event log entry is a block of binary data that is interpreted by the event viewer. The actual size of an event log entry is small. Besides the fixed headers, an entry’s data is less than 256 bytes. The event viewer uses a message catalog to translate the error code in the entry into a locale-specific text message. The text message can include strings that are inserted from the log entry data.

When you write to the event log, consider the following:

* The size of the event log is limited. The administrator of the system determines its size and the rules for overwriting log entries. Given the limited size, make sure that the messages that you write to the event log have a purpose. Administrators do not need to know whether your driver started and stopped correctly.When they check the log, they look for problems.
* APIs exist for reading the event log, and many system administrators have tools that scan for particular errors. Make your driver’s error codes unique, and use a separate code for each error. Also, create only one event per error.An event log entry is not a print statement in which multiple lines are fine.
* Finally, do not create a single global error message and insert text about the actual error from a string.

The limited size of the event log entry influences your design of entries. Unlike internal diagnostics in which text is recommended, event log entries are sometimes better with binary data. Use the unique error code and the text message to give your customer data about decoding the binary.

A final consideration when using the event log is where the message catalog resides. The message catalog is compiled into a resource that is placed in an executable file such as the driver or a DLL. The examples in the WDK show how to include the catalog in your driver. This approach has the limitation that adding or changing messages requires you to rebuild the driver. The other approach is to place the messages into a dummy DLL, which letsyou update the messages without impacting the driver. This makes it possible to distribute the dummy DLL and the message catalog file so that the messages can be internationalized without driver source code.

## Performance Monitoring

A good driver provides feedback to the user through the performance monitor (Sysmon). For common classes of devices, drivers should provide the standard data. Even if your driver does not fit a common category, it should provide monitoring information.

The simplest way to provide the performance monitoring data is through WMI, a powerful instrumentation model that can do much more than support Sysmon. For a complete explanation of WMI, see “Windows Management Instrumentation”in the WDK.

When designing a driver, be sure that it has a WMI class (the container for the data) that meets Sysmon requirements. These requirements include the following items.For an explanation of the individual terms, see the WDK:

* The class must inherit from Win32\_PerfRawData.
* The class must reside in the *\root\cimv2* or *\root\wmi* namespace.
* The Dynamic, Provider(“WmiProv”), and HiPerfclass qualifiers are required, and the Abstract, GenericPerfCtr and Cookedqualifiers are not allowed.
* Each data item must be a signed or unsigned integer of 32-bit or 64-bit size, with CounterType, DefaultScale and Perfdetail qualifiers.

You must also decide which data to provide in the performance monitoring class of your driver. Consider the counters that are available to Sysmon and, in general, think about items such as:

IRP counters.

Provide a count of requests to your driver. Consider both a total and a breakdown by types such as reads, writes, and IOCTLs.

Data counters.

Provide a count of the data that your driver processed.

Failure rates.

Provide the number or rate of failures. This category is important if your device reports failures. Include them in the log.

Security counters.

If your driver has its own security checks, provide a counter of the rejections that occur. You might also want to report failures of requests for invalid data.

A good example of WMI performance data is the Toaster sample in theWDKin the*src\general\toaster* directory.

## Custom Dump Data

A good driver writer considers what is needed from a crash dump to help diagnose a problem. If your device maintains much state information outside the computer’s main memory, you might want to collect this data and add it to the crash dump.

Even if your driver does not collect data from the device for a dump, consider how you analyze the crash dump. Writing a debugger extension to aid in decoding your driver’s data can make crash dumps easier to analyze.

## Version Block

One of the simplest ways to help customers report problems in your driver is to include a version information resource, also called a version block. To create the version information, use a VERSIONINFO resource definition statement in a resource script file. The WDK sample drivers have an RC file with VERSIONINFO, and so should your driver. The information in this block should include your company name, the product name and version, the filename and versions, a description of the file, and a copyright notice.

To display version information in Windows Explorer, right-click a driver file, select **Properties** and,in the **Properties** dialog box, click the **Version** tab to see the version information for that driver.

Your driver should have unique version information for every build. This informationlets customers identify which version of your product they are using if problems occur. They might be using an old version, and you might already have fixed the bug.

Do not just use the resource file (.RC) from a WDK sample driver,which identifies the build revision as “WDK” and the developer as “Microsoft.”Instead,create your own RC file that defines the same fields with data that is appropriate for your company and driver.

A useful approach is to have a version file for the developer’s environment that uses the source control revision, the date, whether the build is checked or free, and the location of the build. Then have a separate controlled file for the release process.

# Test Tools

A high-quality device driver is a well-tested driver. Microsoft provides several testing tools for drivers. These tools test many of the common problem areas for device drivers.

It is not sufficient to run just the Microsoft-supplied tests. Your driver needs additional testing that is specific to its implementation. You can do this testing either by writing custom test code to exercise the driver or by creating scripts to invoke existing tools and utilities. Whichever you choose, consider the following guidelines when you develop and run your tests:

Make the testing easily reproducible.

Testing that requires many manual steps or a specific system is not a viable test. The test should be easy to run and easy to move to additional test machines. If the test requires manual steps, provide a clearly written document that describes what to do.

Test results should be easily verifiable.

Many tests provide detailed logs of their runs. These logs should have a simple result line that indicates whether the tests succeeded. It is unacceptable to have a large log that must be manually verified to determine whether the test ran correctly. At a minimum, create the log so that it can easily be compared to a previous successful log.

Test on various platforms.

This paper has encouraged the use of the checked build of Windows, the checked build of the driver, Driver Verifier, code coverage, and various logging tools. You should run your tests in this environment, but not only in this environment. Using the retail build of Windows and both uniprocessor and multiprocessor systems is required for good testing. Use several systems with different HALs if possible.

When you fix a bug, write a test.

The one thing you should always do when testing is to make sure that your customer never sees the bug again. When you debug a problem, create a regression test that can be integrated into future testing of the driver.

## Device Path Exerciser

Device Path Exerciser (*Devpathexer.exe*) is a powerful testing tool for most drivers. This tool checks the stability of drivers for various types of device IOCTL calls. In addition, the tool provides several options for opening and exercising a device. No driver should ship without a set of tests through Device Path Exerciser. The following are the common tests thatDevice Path Exerciser runs:

* Unexpected I/O requests to the driver.
* Requests with buffers that:

Are missing.

Are too small.

Contain meaningless information.

Contain data that changes asynchronously.

Use invalid pointers.

* Requests in which the mapping of the user buffer changes asynchronously, which causes the pages to become unreadable.
* Relative open operations with arbitrary or hard-to-parse file names and open operations to fictitious device objects.

Previously, Device Path Exerciser required a complex command line. Although the command line is still present for scripting purposes, the current tool has a GUI that you can use for most needs. The command-line interface and documentation for the tool itself are in the WDK documentation.

Be careful to test only your driver. Some drivers do not respond well to the tests, and your test system could see failures that are unrelated to your driver if you choose a broad set of drivers. We recommend that you run the tests initially one at a time, so that you can easily restart the test and debug the problem. Before you ship the driver, you should run it with a broad selection of tests.

The results of a Device Path Exerciserrun appear in several log files. The following table shows a quick summary of the log files.

|  |  |
| --- | --- |
| Dc2.log | This is the general log for the test. It is highly dependent on the specified logging levels (the default is verbose). This log records the start and end of the tests and events in the test. Warning: This log is overwritten eachtime the tool is run. |
| Diags.log | This is the diagnostics messages log. It can record various status values through the tests. The logging level is controllable, with the default being “Verbose”. |
| Crashn.log | This log records data on previously run tests. This file is used to provide information to skip tests. |
| Crash.log | This log records the test that caused the system to crash. The log is created if a skip test option is used. |

Device Path Exerciser is a complex but critical test tool for driver developers.

## IoSpy and IoAttack

IoSpy and IoAttack provide another approach to testing I/O control calls in a driver and testing WMI requests. To use the tools, copy them from the *\tools\iospy\<arch>*directory in the WDK to the test machine. Also, be sure that the file *wttlog.dll* is on the test system.If necessary, copy the DLL from *\bin\<arch>*of the WDK. To run the tool, start by issuing the following command:

iospycmd /D A

The preceding command lists all devices on the system, so you can choose one or more to observe. To begin observation, use the following command:

iospycmd /I <devnode>

In the preceding command,<devnode> is the number that is associated with the device from the previous command. Now exercise the device with any tests that you have that exercise IOCTL and WMI calls. Then disable the observation by using the following command:

iospycmd /R

After you have the observation data, you can run IoAttack. First, you probably want to run the following command:

ioattack /DT

The preceding command lists all tests that will be performed. To run the tests, issue the following command:

ioattack /A

Be sure to have the kernel debugger running before you test your driver so that you can debug any problems that you find. This section provides an overview of these tools. For more information, see the WDK documentation.

## Plug and Play Driver Test

The Plug and Play Driver Test (*pnpdtest.exe*) provides an easy way to test Plug and Play support in your device driver. The test is located in *\tools\pnpd2\<arch>* in the WDK.Copy the directory to the test machine. Also, as with IoSpy,check that*wttlog.dll* is on the test system.If not, copy this DLL from *\bin\<arch>*of the WDK. The test should run cleanly when applied directly to your driver and when invoked for any underlying hardware device in the driver stack. The test has major options that appear in the GUI if you just run the executable, including the following examples:

Removal.

This option attempts to remove the device from the system. Be sure to check that your driver does not just reject all remove requests because the test still passes in this case.

Rebalance.

This test stops and starts the driver to rebalance hardware resources. Note that this test will not run if your driver has no hardware resources. If your driver loads on top of a driver for hardware, apply this test to the device that owns the hardware to test your driver.

Surprise removal.

This test issues a surprise removal to the driver for the device.

Stress.

This test combines rebalance and removal for five minutes, followed by a surprise removal.

The test has several other options that can be invoked only from the command line.The WDK documentation provides a full description.

## PwrTest

One of the most difficult areas to develop correctly in a Windows device driver is power management. The PwrTest tool provides testing in this area.It is located in *tools\powermanagement\<arch>*in the WDK. Pwrtest works by running one of several scenarios.The best approach to obtain data on all scenarios is to issue the following command:

pwrtest /?

pwrtest /<scenario> /?

The first command lists all scenarios, andthe second command displays options for the scenario that you specify. Two of the most common uses of PwrTest are obtaining information and sleep testing a system, as shown in the following commands:

pwrtest /info:all

pwrtest /sleep /c:3 /s:1

The first command displays all information about the system’s power capabilities. The second command runs a sleep test that cycles three times and uses only S1 transitions. You must adjust testing to the needs of your device and consider running a long test to exercise the power capabilities.

## Windows Device Testing Framework

WDTF provides capabilities for running tests and developing scripts of your own. Although developing WDTF tests is beyond the scope of this paper, WDTF provides several packaged scripts that are ready to run for device classes such as networking, audio, video, and USB. These tests are used in the Windows Logo Kit (WLK). The WLK installs WDTF for you.The following section contains more information about the WLK.

To install WDTF, copy the *\WDTF\<arch>*directory from the WDK to your test machine. Then issue the following command:

InstallWDTF /InstallPath<location to install WDTF>

Before you try to run a test, you must get *WTTLogCm.dll* on your test machine and then issue the following command:

Regsvr32 <path to WTTLogCm.dll>

For the Windows Server 2008 WDK,this DLL is located in *\tools\WDTF\<arch>\redist.* Unfortunately for the Windows 7 WDK, the file was accidently omitted and so you need to get the earlier WDK for the DLL.

The following table lists the five sample scripts that are provided with WDTF that you can easily run after you have installed the framework.

|  |  |
| --- | --- |
| EnumDevices.wsf | Enumerates all devices on the system. |
| Basic\_SimpleIO.wsf | Runs simple I/O stress tests. |
| Disable\_Enable\_With\_IO.wsf | Runs simple I/O stress tests and Plug and Play stress tests. |
| Sleep\_Stress\_With\_IO.wsf | Runs simple I/O stress tests and power management stress tests. |
| Common\_Scenario\_Stress\_With\_IO.wsf | Runs simple I/O stress tests, Plug and Play stress tests, and power management stress tests. |

The WDK documentation covers the options and capabilities of these scripts in detail, as well as how to program your own scripts.

## Windows Logo Kit

The WLKis a set of tests that hardware and software must pass to qualify for the Windows Logo Program. The primary goal of the WLK is to test hardware and driver compatibility with a known standard for a given class of device. The logo kit tests many classes of devices.

The challenge of the WLK is that it requires a complex environment that consists of at least three systems to run it. Because of its complexity and because many of the tests involve tools that were listed earlier in this paper, you should defer testing with the WLK until after the other tests run cleanly. However, your driver should pass the logo tests before it is released. For more information about the WLK and how to get it, see the “Windows Logo Program” section of the WHDC website.

# Summary

Driver development is a complex task, and producing a high-quality driver requires careful attention to detail. Windows provides many tools to identify problems throughout the development cycle. To find bugs early, you should take the time to use these tools. Finding the bugs early is the best way to reduce the overall costs of the driver. In general, you should:

Use the latest WDK Windows build environment.

The latest WDK provides the newest samples, tools, and documentation. Using anything other than the standard WDK build environment can cause subtle problems and make it more difficult to use some tools.

Find bugs at compile time.

Using the compiler with /W4 /Wp64, C++, PREfast,and SDV to find problems makes your debugging easier. Enabling DEPRECATE\_DDK\_FUNCTIONS and C\_ASSERT protects you from future problems. Checking INF files with ChkINF is the easiest way to obtain a clean installation.

Runtime checks and logging are debugging tools.

Driver Verifier, Call Usage Verifier, and the checked build of Windows are debugging tools. Using pool tagging with all checking enabled helps find leaks and problems. Your driver should run cleanly with all these tools. Profiling your driver for code coverage and performance should be a standard step in the development process. Monitoring the various logging capabilities of the system can help find otherwise overlooked problems in your driver.

Planning diagnostics for maintenance makes development and support easier.

Liberal use of diagnostics such as debug prints, WPP tracing, and assertions enables quick isolation of bugs in your code. Event logging, performance monitor support, and a version block help your customers provide good feedback when problems occur.

Take advantage of the tests that Microsoft provides.

The test tools in the WDK are a good base for your testing. Be sure to include your own tests to exercise your driver. No one can ever test too much.

# Resources

#### White Papers and Articles

Driver Development on WHDC

Driver Lifecycle Fundamentals: Overview  
<http://www.microsoft.com/whdc/driver/default.mspx>

Driver SigningRequirements for Windows  
<http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx>

Windows Driver Kit (WDK) Build Environment  
<http://www.microsoft.com/whdc/DevTools/WDK/WDK_Build.mspx>

C++ for Kernel Mode Drivers: Pros and Cons  
<http://www.microsoft.com/whdc/driver/kernel/KMcode.mspx>

PREfast Step-by-Step  
<http://www.microsoft.com/whdc/DevTools/tools/PREfast_steps.mspx>

PREfast Annotations  
<http://www.microsoft.com/whdc/DevTools/tools/annotations.mspx>

Debugging Device Installation in Windows Vista  
<http://www.microsoft.com/whdc/driver/install/diagnose.mspx>

Troubleshooting Device Installation with the SetupAPI Log File  
<http://www.microsoft.com/whdc/driver/install/setupapilog.mspx>

Microsoft Knowledge Base

How to enable verbose debug tracing in various drivers and subsystems  
<http://support.microsoft.com/default.aspx?scid=kb;en-us;314743>

#### Driver Development Kits and Tools from Microsoft

Windows Driver Kit

How to Get the WDK  
<http://www.microsoft.com/whdc/DevTools/WDK/WDKpkg.mspx>

Debugging Tools for Windows - Overview

WinDbg and other kernel debuggers, extensions, and tools  
<http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx>

Windows SDK

<http://msdn.microsoft.com/en-us/windows/bb980924.aspx>

Windows Logo Kit: Overview

Numerous papers on the WLK <http://www.microsoft.com/whdc/winlogo/default.mspx>

Windows Logo Program Test Categories

<http://www.microsoft.com/whdc/winlogo/categories.mspx>

Microsoft Developer Network

Checked builds of Windows and other developer resources  
<http://msdn.microsoft.com>

#### Third-Party Tools for Driver Development

Code Coverage Tools

BullseyeCoverage  
<http://www.bullseye.com>

DDKbuild batch build

Hollis Technology Solutions  
<http://www.hollistech.com> (click **Resources**)

DDKBUILD -- Visual Studio .CMD Procedure For Building Drivers

Open Systems Resources  
<http://www.osronline.com/article.cfm?article=43>

DeviceTree

Open Systems Resources  
<http://www.osronline.com/article.cfm?article=97>

Tools for setting up test systems

OSRSYSTOOL Osr Test System Configuration Utility  
<http://www.osronline.com/article.cfm?article=537>

BellaVista  
<http://www.zezula.net>

Tools for monitoring debug print messages

DebugMon   
<http://www.osronline.com/article.cfm?article=99>

DebugView for Windows v4.76  
<http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx>