Tracing and Diagnosability for WDF Drivers

October 19, 2008

Abstract

Software tracing for drivers is usually based on Event Tracing for Windows® (ETW), a kernel-level facility that logs trace messages for both kernel-mode and user-mode processes. Because ETW can be somewhat complicated to use, most driver developers use the Windows software trace preprocessor (WPP), which simplifies and enhances the process of instrumenting a driver for ETW tracing. This paper discusses how to implement WPP tracing in a Windows Driver Foundation (WDF) driver.

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

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

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

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

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

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

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

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

© 2008 Microsoft Corporation. All rights reserved.

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

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

Contents

[WPP Software Tracing Basics 4](#_Toc208806371)

[Advantages of WPP Software Tracing 4](#_Toc208806372)

[WPP Software Tracing Components 6](#_Toc208806373)

[WPP and ETW 8](#_Toc208806374)

[ETW in Windows Vista 8](#_Toc208806375)

[Trace Message Functions and Macros 9](#_Toc208806376)

[DoTraceMessage Macro 9](#_Toc208806377)

[Message Conditions 10](#_Toc208806378)

[How to Convert Standard Debug Print Statements to ETW 11](#_Toc208806379)

[Custom Trace Message Functions 12](#_Toc208806380)

[How to Support Software Tracing in a Driver 12](#_Toc208806381)

[Modify Sources to Run the WPP Preprocessor 13](#_Toc208806382)

[UMDF Example: RUN\_WPP for Fx2\_Driver 14](#_Toc208806383)

[KMDF Example: RUN\_WPP for the Osrusbfx2 Sample 15](#_Toc208806384)

[Include the TMH File 16](#_Toc208806385)

[UMDF Example: Including the TMH File 16](#_Toc208806386)

[KMDF Example: Including the TMH File 16](#_Toc208806387)

[Define the Control GUID and Trace Flags 17](#_Toc208806388)

[UMDF Example: Defining the Control GUID and Trace Flags 18](#_Toc208806389)

[KMDF Example: Defining the Control GUID and Trace Flags 18](#_Toc208806390)

[Initialize and Clean Up Tracing 19](#_Toc208806391)

[Initialize Tracing 19](#_Toc208806392)

[UMDF Example: Initialize Tracing in DllMain 19](#_Toc208806393)

[KMDF Example: Initializing Tracing in DriverEntry 20](#_Toc208806394)

[Clean Up Tracing 20](#_Toc208806395)

[UMDF Example: Clean Up Tracing in DllMain 20](#_Toc208806396)

[KMDF Example: Cleaning up Tracing in *EvtCleanupCallback* 21](#_Toc208806397)

[Instrument the Driver Code 22](#_Toc208806398)

[UMDF Example: Adding Trace Message Calls to Driver Code 22](#_Toc208806399)

[KMDF Example: Adding Trace Message Calls to Driver Code 22](#_Toc208806400)

[Tools for Software Tracing 23](#_Toc208806401)

[How to Run a Software Trace Session 24](#_Toc208806402)

[Prepare the Driver 24](#_Toc208806403)

[View a Trace Log by Using TraceView 25](#_Toc208806404)

[How to Create and View a Trace Log File 25](#_Toc208806405)

[How to View a Trace Log in Real Time 26](#_Toc208806406)

[View Trace Messages in Real Time with WinDbg 27](#_Toc208806407)

[View the Frameworks Trace Log by Using the Core Tracing Tools 28](#_Toc208806408)

[Best Practices: Design for Diagnosability 30](#_Toc208806409)

[Resources 30](#_Toc208806410)

# WPP Software Tracing Basics

Software tracing is one of the oldest debugging techniques and provides a convenient and flexible way to generate detailed information about the behavior of a running driver at key points in its operation. It consists of embedding trace message statements in the driver code to record its behavior at various points. Tracing is typically used to determine the location and cause of bugs, but it can also be used for such tasks as profiling how frequently different routines are called.

Software tracing creates a running log of the driver’s operation. You can use trace messages to put a variety of useful information into the trace log, such as:

* Configuration settings.
* The cause of state changes.
* Notification of scavenging or cleanup operations.
* Memory allocations.
* I/O activities, such as request completion or request cancellation.
* Error data.

Software tracing for drivers is usually based on Event Tracing for Windows®bb (ETW), a kernel-level facility that logs trace messages for both kernel-mode and user-mode processes. The messages can be viewed in real time—while the driver is in operation—or retrieved later from log files. Because ETW can be somewhat complicated to use, most driver developers use the Windows software trace preprocessor (WPP), which simplifies and enhances the process of instrumenting a driver for ETW tracing.

This paper discusses how to implement WPP tracing in a WDF driver, using as examples the two WDF drivers for the OSR USB Fx2 learning kit.

* Fx2\_Driver is a UMDF version of the driver. It is located under the WDK root folder at: *VersionNumber*\src\umdf\usb\fx2\_driver.
* OsrUsbFx2 is a KMDF version of the driver. It is located under the WDK root folder at: *VersionNumber*\src\kmdf\osrusbfx2\sys.

Both samples consist of a series of increasingly full-featured versions of the driver, each with its own folder. This paper is based on the final version of the driver, which is located in the “final” folder.

We recommend that you read both sets of examples, even if you are only interested in one of the frameworks. The two samples illustrate somewhat different approaches to implementing WPP tracing, and most of the discussion applies to either framework.

## Advantages of WPP Software Tracing

Software tracing can be done in many ways. WPP software tracing provides the following significant advantages over techniques, such as debug print statements:

Dynamic and Flexible Control

You can enable or disable message generation while the driver is running—you are not required to stop and restart the driver or reboot the operating system. You can flag trace messages so that you can choose to observe only a selected subset of the available messages. In addition, you can include messages from multiple sources in the same trace log. For example, you could choose to view messages from a driver plus related operating system components.

Ability to View Messages in Real Time or to Store in a File

In Windows XP and later versions of Windows, you can view WPP trace messages in real time. You can also direct the messages to a trace log file and view them later, perhaps on a different computer.

Rich Information

WPP trace messages can include any data you find useful, such as the current driver configuration, error codes, or the values of key variables. The WPP preprocessor adds the function name, source file name, and line number to each trace message when the project is built. When a trace message is issued, the ETW logging mechanism automatically attaches a time stamp.

Intellectual Property Protection

The driver issues WPP trace messages in binary format. You must have the associated formatting information to make the messages readable. That information is stored separately from the driver binary and is not usually shipped with the product. This feature makes it difficult for an unauthorized person to obtain intellectual property from trace messages or use them to reverse-engineer the driver.

Easy Migration from Debug Print Statements

If a driver currently uses debug print statements for tracing, you can direct WPP to convert the debug print calls to trace messages during the build process. This process converts your existing trace instrumentation to a more efficient form without rewriting your code.

Inclusion in Shipped Products

WPP software tracing can be included with both checked and free builds of a driver, so product binaries can ship with the tracing code in place. You can use WDK tracing tools in the field to enable and view trace messages on drivers that are installed on customer systems.

Minimal Performance Impact

WPP trace message calls are issued only when tracing is explicitly enabled by an external controller. When the driver is compiled, the WPP preprocessor converts the trace message in your code to the following form:

if (tracing\_is\_enabled) {

//Call the trace message routine

}

If tracing is not enabled, the trace message routine is never called and the performance impact is negligible. Even when tracing is enabled, the performance impact on the driver is minor because trace messages are issued in binary format. A separate application handles the time-consuming task of formatting and displaying the messages.

Tracing is especially useful for investigating bugs—sometimes called “heisenbugs”—whose behavior is very sensitive to system performance. These bugs often disappear when you attempt to observe driver behavior by using techniques, such as debug print statements, which can slow the driver to the point that the bug does not manifest itself.

## WPP Software Tracing Components

WPP software tracing involves several components that work together in a trace session:

Trace Provider

A trace provider is an application, operating system component, or driver that is instrumented for tracing. Usually, a driver represents itself as a single trace provider. However, a driver can be separated into multiple WPP trace providers, even within a single source file. This paper discusses how to instrument a WDF driver as a single trace provider.

Trace Controller

A trace controller is an application or tool that manages a trace session. A trace provider generates trace messages only if it is enabled by a trace controller. Two commonly used trace controllers—TraceView and Tracelog—are included with the WDK tools.

A trace controller can enable multiple providers for a session, including providers from different drivers or system components. When a session has multiple providers, the messages are interleaved in the trace log in the order in which they are received. If a driver defines multiple providers, the trace controller can be configured to enable messages only from selected providers. However, with WPP tracing, different providers from the same driver cannot be enabled simultaneously in more than one session. A driver can contribute trace messages to only one session at any given time, regardless of the number of providers it defines.

See “Controlling Event Tracing Sessions” in the Platform SDK for a discussion of how to write your own trace controller by using the ETW API.

Trace Buffer

The system maintains a set of buffers to store trace messages for each trace session. A trace controller can configure the size of the buffers for a session. Buffers are automatically flushed to the trace log file or trace consumer at a specified interval or when the buffers become full, whichever happens first. Buffers are flushed automatically when the trace controller stops the trace session. However, the trace controller can also explicitly flush the buffers, either on demand or at regular intervals.

Trace Session

A trace session is a time period during which one or more trace providers have been enabled and generate a series of trace messages that are written to a trace buffer. This series of messages is called a trace log. A trace controller starts and configures the session, enables one or more providers, and associates the providers with the session. During a session, the trace controller can query and update properties of the trace session and, finally, stop the session.

If a trace session is associated with the Windows kernel logger, the trace log includes messages associated with predefined system events, such as disk I/O or page faults. The system trace messages are interleaved with driver messages in the order that they are generated, so the trace log captures the actions of the driver in relation to key Windows kernel events.

With WPP, a trace provider can be associated with only one trace session. If a controller enables that provider in another session, the provider is dropped from the original session.

Trace Consumer

A trace consumer is an application that receives, formats, and displays a session’s trace log, either in real time or from a log file. The trace consumer formats trace messages in human-readable format by using instructions either from the provider’s program database (.pdb) symbol file or from a separate trace message format (.tmf) file. Some trace consumers also function as a trace controller.

Figure 1 shows how trace consumers interact with the other trace components on a typical system.



Figure 1. Software tracing architecture

Figure 1 shows three active trace sessions and two trace consumers:

* Session 1 has no providers or consumers currently associated with it.

A session can exist without either providers or consumers.

* Session 2 has two associated providers: A and B.

The trace controller transfers trace messages from the trace session buffers directly to Consumer 1, which displays the interleaved messages from both providers in real time.

* Session 3 has a single provider: C.

The trace controller writes the messages to a trace log(.etl)filefor that session. They are displayed by Consumer 2 after the session is finished, perhaps on a different computer.

## WPP and ETW

WPP software tracing is essentially an abstraction layer over ETW. It simplifies and enhances ETW tracing, especially for drivers that need only lightweight tracing. WPP works in the following way:

1. A driver includes WPP macros in its source code to configure tracing and to generate trace messages.

2. The project's *Sources* file directs the Build utility to run the WPP preprocessor before the first compiler pass. The preprocessor converts the WPP macros and template files to the code that is required to implement ETW tracing.

3. The WPP-generated code for each source file is stored in an associated trace message header(.tmh)file. This file also contains macros that add instructions for formatting trace messages to the driver's PDB file.

4. Each .c or .cpp file includes its associated TMH file. When the source code is compiled, the code in the TMH file is compiled along with it, just like a conventional header file.

## ETW in Windows Vista

Beginning with Windows Vista, Windows supports a unified event format that allows applications to use the Windows Vista ETW API to log schematized events to ETW. The event format uses an extensible markup language (XML) manifest file that is later compiled as a resource and attached to the driver binary. Each event has standard metadata information and a variable payload section that can contain a set of simple types or data structures, such as arrays or structures. This information allows trace message consumers to decode the event after it is logged to ETW.

**Note:** WPP does not use the Windows Vista ETW API, but instead logs events by using the earlier version of the ETW API. To use the new Windows Vista features, you must implement tracing with the Windows Vista ETW API.

Logging events in the Windows Vista format is useful for components that use the event logs. Components, such as the Event Log service and the Event Viewer can easily use the schematized events to implement discoverability, querying, and filtering capabilities that allow users to quickly focus on the events of interest. The format also allows event and other messages to be localized. In addition, “self-healing” components like Windows Diagnostic Infrastructure use the schematized event format to monitor for events associated with specific problems. When such an event occurs, if a resolution module is specified, Windows Diagnostic Infrastructure can automatically fix the problem.

See “Adding Event Tracing to Kernel-Mode Drivers” in the WDK for a discussion of the Windows Vista ETW API. For an example that shows how to implement tracing by using the Windows Vista ETW API, see the Evntdrv sample in the WDK at WinDDK\*BuildNumber*\src\general\evntdrv.

# Trace Message Functions and Macros

The core of any tracing implementation is the function that generates the trace message. You insert trace message function calls at appropriate points in your code and configure the function to place relevant information into the message. Typically, a trace message consists of some literal text to define the message context plus the values of one or more interesting variables.

Trace message functions typically also support one or more conditions, which can be used to categorize the message. You can, for example, use a condition to assign different levels of severity to a trace message. Trace controllers can selectively enable conditions to direct the provider to generate only a limited subset of messages, such as those reporting warning or error conditions.

WPP trace messages can be generated in several ways:

* By using the default **DoTraceMessage** macro.

The WPP preprocessor converts **DoTraceMessage** into the appropriate ETW function call. The ETW function varies depending on the version of Windows for which the driver is compiled and whether the driver runs in user mode or kernel mode.

* By converting debug print functions to trace message functions.

You can direct the WPP preprocessor to convert existing debug print statements to equivalent WPP trace message functions.

* By using custom trace message functions.

These functions are useful if **DoTraceMessage** does not support your driver's requirements. Examples of custom trace message functions appear later in this paper.

**Important:** Be careful about exposing security or privacy data in trace messages. Any trace consumer that can access your driver’s TMF files or PDB file can format and read your driver's trace log.

## DoTraceMessage Macro

The DoTraceMessage macro is the default way to generate trace messages. It has the following format:

**DoTraceMessage** (*Flag*, ”*Message*”, *MessageVariables*...);

The macro’s parameters are:

*Flag*

A 32-bit value that contains a set of user-defined trace flags. Trace consumers use these flags to specify which categories of trace message to generate. For examples of how to define and use these flags, see “How to Support Software Tracing in a Driver” later in this paper.

*Message*

A string constant that defines the format of the trace message. The string is similar to the **printf** format string and usually includes literal text interspersed with standard **printf** format specifications, such as %d and %s. WPP also supports some special formats, such as %!NTSTATUS! and %!HRESULT!, that display string equivalents of kernel-mode status values and user-mode HRESULT values, respectively.

*MessageVariables*

A comma-delimited list of variables whose values are added to the message string according to the format specified by *Message*.

The WPP preprocessor converts **DoTraceMessage** to the following form:

if(Flag is enabled by the trace controller)

{

//Call the appropriate ETW tracing function

}

See “**DoTraceMessage**”in the WDK for a detailed description.

## Message Conditions

Most trace message functions allow you to set one or more conditions on the message. For example, you can use a condition to specify whether a trace message is simply informational or is associated with a warning or error condition. The trace controller can then selectively enable conditions to limit the messages in the trace log to certain categories.

There are two standard sets of conditions:

* FLAGS is a 32-bit value that takes a set of as many as 32 user-defined flag values. You specify the available values when you define the RUN\_WPP directive, which is discussed in “How to Support Software Tracing in a Driver,” later in this paper. **DoTraceMessage** uses the FLAGS condition.
* LEVEL is 1-byte value that is used by some debug print functions and custom trace message functions, typically to indicate the seriousness of the event associated with the message. It can take any user-defined value, but is commonly set to one of the trace levels declared in Evntrace.h. Table 1 lists these trace levels.

Table 1. Trace Levels

|  |  |  |
| --- | --- | --- |
| Condition | Value | Description |
| TRACE\_LEVEL\_NONE | 0 | Tracing is not on. |
| TRACE\_LEVEL\_CRITICAL | 1 | Abnormal exit or termination. |
| TRACE\_LEVEL\_ERROR | 2 | Severe errors. |
| TRACE\_LEVEL\_WARNING | 3 | Warnings, such as allocation failure. |
| TRACE\_LEVEL\_INFORMATION | 4 | Informational messages, including non-error cases, such as entry or exit. |
| TRACE\_LEVEL\_VERBOSE | 5 | Detailed information from intermediate steps. |

Conditions are used to categorize messages. A trace controller can then configure a session by selectively enabling conditions to direct the session’s trace providers to generate messages only from selected categories. This feature allows a trace controller to increase the signal-to-noise ratio of the trace log by restricting it to only a selected subset of trace messages.

For example, if a message condition takes the standard trace levels, a trace consumer can set LEVEL conditions to view only error and warning messages and ignore informational and verbose messages. A trace message function could also take a FLAGS condition that indicates which part of the driver generated the message. A trace consumer could then set the LEVEL and FLAGS conditions, for example, to view only those error messages that are associated with device I/O control requests.

**DoTraceMessage** has a single condition, which takes user-defined message flags. You can also implement a custom trace message function that uses a different condition or multiple conditions.

## How to Convert Standard Debug Print Statements to ETW

Many drivers already use standard WDK debug print functions, such as **DebugPrintEx** or **KdPrintEx**. If your code already contains such functions, you can direct WPP to convert them to equivalent trace message macros by including the following option in the RUN\_WPP directive in your project’s *Sources* file:

**-func:***DebugPrintFunction*(*Condition1*,...,MSG,...)

*DebugPrintFunction* is the name of your debug print function, and *Condition1* and so on are the conditions that can be set for the debug print function.

For example, the simplest debug print function is **printf**. **printf** has no conditions, so to map **printf** to an equivalent WPP trace message function, use the **–func** option in the following form:

**-func:**printf**{LEVEL=***Flag***}**(MSG,...)

The {LEVEL=*Flag*} specification globally applies a condition to all occurrences of **printf.** If you omit the {LEVEL=*Flag*} specification altogether, the debug print function always executes, even when the trace provider is not enabled. This directive creates a WPP macro that has the LEVEL condition, uses the format string as the MSG parameter, and inserts v1, v2, and so on, into the message as specified by MSG. The message will be similar to what would have been printed, except that it is now sent to a trace log. *Flag* indicates the level at which tracing is enabled. The *Flag* value is one of the flags that are defined in the WPP\_DEFINE\_BIT element of the [WPP\_CONTROL\_GUIDS](http://msdn.microsoft.com/en-us/library/ms793151.aspx) macro. If you do not define custom flags for your driver, WPP defines three default flags: Error, Unusual, and Noise. Their values are 1, 2, and 4, respectively. Thus, the following option enables a **printf** trace message for errors at Noise level or greater:

-func:printf{LEVEL=Noise}(MSG,...)

A more commonly used function is **DebugPrintEx**, which has two conditions, followed by a format string and a list of variables. To map **DebugPrintEx** to an equivalent WPP macro, use:

-func:DebugPrintEx(FLAG, LEVEL, MSG,...)

You can convert **KdPrintEx** to an equivalent WPP macro in much the same way, but you must use double parentheses, as follows:

–func:KdPrintEx(LEVEL,(MSG,…))

In this example, the resulting WPP trace message is enabled or disabled based only on the LEVEL value.

You must also define custom WPP\_*CONDITIONS\_*ENABLED and WPP\_*CONDITIONS\_*LOGGER macros that correspond to your conditions. For a description of how to define these macros, see “UMDF Example: RUN\_WPP for Fx2\_Driver”, later in this paper. For more information on custom trace messages, see the WDK documentation.

## Custom Trace Message Functions

If **DoTraceMessage** or standard debug print functions do not meet the requirements of your driver, you can define a custom trace message function. A custom trace message function must follow a similar general format to that used by **DoTraceMessage**. The difference is that custom trace message functions can support different conditions, including multiple conditions, as shown in the following example:

FunctionName (Condition1,

Condition2,

...**,**

"Message"**,**

MessageVariables...)

The parameters for custom trace message functions are as follows:

*Conditions*

A comma-delimited list of one or more conditions. The trace message is generated only if all conditions are true. Custom trace message functions typically use only one or two conditions, FLAGS and/or LEVEL.

*Message*

A string constant that defines the format of the trace message. The format specifications are the same as those for **DoTraceMessage**.

*MessageVariables*

A comma-separated list of driver-defined variables. The values of the variables are added to the message according to the format specified by the *Message* argument, much as they are with **printf**.

To use a custom trace message function, you must configure your project so that the WPP processor recognizes the custom function and correctly handles the output. The procedure is similar to that used to convert standard debug print functions. See “UMDF Example: RUN\_WPP for the Fx2\_Driver Sample” later in this paper for an example of how to implement a custom trace message function.

# How to Support Software Tracing in a Driver

To instrument a driver for software tracing, you must:

* Modify the *Sources* file to run the WPP preprocessor.
* Include the associated TMH file in your C or C++ source files.
* Define the control GUID and trace flags.
* Add macros to the appropriate callback functions to initialize and clean up tracing.
* Instrument the driver code to generate trace messages at appropriate points.

This section examines the mechanics of implementing WPP software tracing in a WDF driver. Both UMDF and KMDF drivers require essentially the same procedures, although some of the WPP macros are defined differently for user mode than for kernel mode.

## Modify Sources to Run the WPP Preprocessor

WPP is integrated into the WDK build environment, but you must explicitly direct the WDK build utility to run the WPP preprocessor. To do so, add a RUN\_WPP directive to the end of the driver’s *Sources* file. This directive can also convert debug print statements to WPP trace messages and to specify custom trace message functions. A RUN\_WPP directive has the following form:

RUN\_WPP = $(SOURCES) Option1 Option2 ...

The options you use depend on whether you are building a UMDF or a KMDF driver and how you have implemented tracing in your driver. Table 2 summarizes the RUN\_WPP options that are most useful for WDF drivers.

Table 2. RUN\_WPP Options for Drivers

|  |  |
| --- | --- |
| Option | Description |
| **-km** | Required for KMDF drivers; not used for UMDF drivers. By default, only user-mode components are traced. This option defines the WPP\_KERNEL\_MODE macro, which traces kernel-mode components. |
| **-func:***funcname* (*param1*, *param2*, …) | Optional. Defines a custom trace function or a debug print function to be converted. You can repeat this argument to define or convert more than one such function. |
| **-scan:***filename* | Optional. Scans *filename* to find the definition of a custom trace function. The format for the definition appears in the next section of this paper. |
| **-dll** | Required for UMDF drivers; not used for KMDF drivers. Defines the WPP\_DLL macro, which causes the WPP data structures to be initialized whenever WPP\_INIT\_TRACING is called. Otherwise, the structures are initialized only once. |
| **-gen:{***filename.tpl***}**\*.tmh | Optional. Directs the WPP preprocessor to create a TMH file from a template named *filename.tpl*. |

**Note:** The -**func** and **-scan** options are used only for projects that have one or more custom trace message functions, or that are converting debug print functions to WPP macros. If you use only **DoTraceMessage** to generate trace messages, omit these options. If you define a custom trace message function or convert your debug print functions, **-func** and **-scan** are optional only to the extent that you must use one or the other to provide WPP with the function prototype. See the following two sections for examples of how to use the **-func** and **-scan** options.

See “WPP Software Tracing” in the WDK for more details about RUN\_WPP options.

### UMDF Example: RUN\_WPP for Fx2\_Driver

The following example shows the RUN\_WPP directive from the Fx2\_Driver sample's *Sources* file:

RUN\_WPP= $(SOURCES) -dll -scan:internal.h

Because this is a UMDF driver, the **-dll** option must be set. Fx2\_Driver defines a custom trace message function in Internal.h, so RUN\_WPP uses the **-scan** option to direct the WPP preprocessor to Internal.h for the custom function definition. The following sample shows the function definition from Internal.h:

// begin\_wpp config

// FUNC TraceEvents(LEVEL, FLAGS, MSG, ...);

// end\_wpp

Custom trace message function definitions in header files have the following characteristics:

* All elements of the definition are commented to prevent the compiler from trying to parse them.
* The definition begins with **begin\_wpp** **config** and ends with **end\_wpp** statements.
* Each function definition is preceded by the **FUNC** keyword, followed by the function name.
* The function name is followed by the input parameters.

If you want to set one of the conditions globally, you can use the following syntax:

// begin\_wpp config

// FUNC TraceEvents{FLAG=MYDRIVER\_ALL\_INFO}(LEVEL, FLAGS, MSG, ...);

// end\_wpp

This sets the TraceEvents FLAG value to MYDRIVER\_ALL\_INFO for all instances of the function.

You can also put options in an .ini file and use **scan** to reference the file. In that case, the content of the file is simply appended to the RUN\_WPP directive, so you use the same syntax that you would use to add the options directly to RUN\_WPP.

See “Can I Customize **DoTraceMessage**?” in the WDK for details about the format of trace message definitions.

The TraceEvents function in Fx2\_Driver supports two conditions: LEVEL and FLAGS. LEVEL takes the standard trace levels that are defined in Evntrace.h. FLAGS has four user-defined values that indicate which driver component generated the message. The FLAG values are defined in the WPP\_CONTROL\_GUIDS macro, as discussed in “Define the Control GUID and Trace Flags” later in this paper.

Each trace message function that you use must have a corresponding pair of macros, as described in Table 3.

Table 3. Standard WPP Macros

|  |  |
| --- | --- |
| Macro | Description |
| WPP\_*CONDITIONS*\_LOGGER | Used to find the trace session associated with the provider and returns a handle to the session. |
| WPP\_*CONDITIONS*\_ENABLED | Used to determine whether logging is enabled with the specified conditions. |

*CONDITIONS* is a list of the conditions that the function supports, in the order in which they are declared in the function’s parameter list, separated by underscore characters. The default trace message function, **DoTraceMessage**, supports the LEVEL condition, so WPP\_LEVEL\_ENABLED and WPP\_LEVEL\_LOGGER are defined by default.

You must define custom versions of the two macros if you do either of the following:

* Define a custom trace message function that uses conditions other than LEVEL.
* Convert a debug print function that uses conditions other than LEVEL.

For example, if a custom trace message function or debug print function supports the LEVEL and FLAGS conditions, you must define WPP\_LEVEL\_FLAGS\_ENABLED and WPP\_LEVEL\_FLAGS\_LOGGER. You typically define custom versions of these macros in terms of the default versions, WPP\_LEVEL\_ENABLED and WPP\_LEVEL\_LOGGER.

* If a driver contains multiple providers, WPP\_*CONDITIONS*\_LOGGER is used to ensure that the appropriate handle is chosen for a particular provider. Because drivers usually only contain a single provider, most trace message functions can just use the default logger by setting WPP\_*CONDITIONS*\_LOGGER to WPP\_LEVEL\_LOGGER.
* You must usually provide a substantive definition for WPP\_*CONDITIONS*\_ENABLED.

The Fx2\_Driver sample defines the macros for TraceEvents in Internal.h, as shown in the following example:

#define WPP\_LEVEL\_FLAGS\_LOGGER(lvl,flags) \

WPP\_LEVEL\_LOGGER(flags)

#define WPP\_LEVEL\_FLAGS\_ENABLED(lvl, flags) \

(WPP\_LEVEL\_ENABLED(flags) && WPP\_CONTROL(WPP\_BIT\_

## flags).Level >= lvl)

The macros define the order of the TraceEvents parameters are LEVEL, followed by FLAGS. The macros are defined as follows:

* WPP\_LEVEL\_FLAGS\_LOGGER sets the macro to WPP\_LEVEL\_LOGGER.
* WPP\_LEVEL\_FLAGS\_ENABLED returns TRUE if logging is enabled for the specified FLAGS value and the enabled LEVEL value is greater than or equal to the level argument.

### KMDF Example: RUN\_WPP for the Osrusbfx2 Sample

The Osrusbfx2 sample also uses the custom TraceEvents trace message function. The message is discussed in detail in the previous section. The following example shows the RUN\_WPP directive from the Osrusbfx2 sample’s *Sources* file.

RUN\_WPP= $(SOURCES) \

-km \

-func:TraceEvents(LEVEL,FLAGS,MSG,...) \

This example uses a continuation character (“\”) to break the directive across several lines. The directive includes two options:

* **–km** indicates that the driver runs in kernel mode.
* **–func** specifies one custom trace message function—TraceEvents—that supports LEVEL and FLAGS conditions.

The Osrusbfx2 source code also includes a definition of TraceEvents in the Trace.h header file. Tracing is enabled for Osrusbfx2 only if EVENT\_TRACING is defined as TRUE. The TraceEvents definition is in a block of code that is conditionally compiled only when EVENT\_TRACING is not set to TRUE. The purpose is to provide a default definition for the TraceEvents function when tracing is not enabled and the WPP preprocessor does not produce a TraceEvents function.

The *Sources* file also defines custom versions of WPP\_LEVEL\_LOGGER and WPP\_LEVEL\_ENABLED, as discussed earlier.

## Include the TMH File

When you run the WPP preprocessor, it creates a .tmh file for each C/C++ source file that contains WPP macros. The TMH file contains the code that the WPP preprocessor generates and must be compiled with the associated source file. Every source file that contains WPP macros must include its corresponding TMH file, which has the same name as the source file, but with a .tmh extension. For example, the TMH file for Driver.c is Driver.tmh and the **#include** directive is as follows:

#include driver.tmh

...

The TMH file must be included after the WPP\_CONTROL\_GUIDS macro and before any other WPP macro calls. WPP\_CONTROL\_GUIDS is typically declared in a common header file, so you must include the TMH file after the common header file.

### UMDF Example: Including the TMH File

In the Fx2\_Driver sample, the WPP\_CONTROL\_GUIDS macro is in the common header file Internal.h. To ensure that WPP\_CONTROL\_GUIDS comes first, the source files include Internal.h before the TMH file. For example, ReadWriteQueue.cpp includes the files as follows:

#include "internal.h"

#include "ReadWriteQueue.tmh"

### KMDF Example: Including the TMH File

The Osrusbfx2 driver defines the control GUID and trace flags in a common header file, Trace.h, which in turn is included in Osrusbfx2.h. Consequently, the source files for this driver include Osrusbfx2.h before the TMH file, as Interrupt.c does in the following example:

#include <osrusbfx2.h>

#if defined(EVENT\_TRACING)

#include "interrupt.tmh"

#endif

The KMDF sample includes the TMH file only if event tracing is enabled. The sample defines the EVENT\_TRACING environment variable in the *Sources* file along with the ENABLE\_EVENT\_TRACING variable. When ENABLE\_EVENT\_TRACING is defined, the *Sources* file also defines EVENT\_TRACING and includes the RUN\_WPP directive, which invokes the WPP preprocessor. This technique allows you to include or exclude WPP tracing at build time.

## Define the Control GUID and Trace Flags

Each trace message provider must have a control GUID that trace controllers use to identify the driver as a trace message provider. Typically, each driver has a separate control GUID. However, a one-to-one relationship between control GUID and driver is not required because:

* A driver can have multiple control GUIDs.

Each GUID must identify a unique trace message provider, and the recommended practice is for drivers to contain a single provider. However, a driver can be broken into multiple providers, each with its own GUID. For example, if a driver has a shared library, the driver could define two control GUIDS: one for the driver and one for the library. Each provider must have its own definition of the WPP\_DEFINE\_CONTROL\_GUID macro and must define its own set of trace flags. The trace consumer can choose to enable either or both control GUIDS as trace message providers for the session.

* Multiple drivers can use the same control GUID.

All components that use the same control GUID operate as a single provider. It is even possible to assign the same control GUID to a combination of kernel-mode and user-mode drivers. If the provider is enabled, the trace messages from every component that shares the GUID are interleaved in the trace log in the order in which they are generated. This approach can be useful for handling drivers that interact during execution, such as a function driver and a related filter driver or a hybrid WDF driver with both UMDF and KMDF components.

To create a GUID, use the UuidGen tool from the Platform SDK to generate a new control GUID specifically for your trace provider. Do not copy the examples in this book or use one generated for another purpose, such as identifying a device interface. For details, see “Using UUIDGEN.EXE” in the MSDN library.

The WPP\_CONTROL\_GUIDS macro specifies the control GUID and the trace flags for a provider. You must define WPP\_CONTROL\_GUIDS for each source file that generates trace messages, typically in a common header file. Each flag has a name and a value that specifies the bit with which the flag is associated. The following shows the general form of the macro definition:

#define WPP\_CONTROL\_GUIDS \

WPP\_DEFINE\_CONTROL\_GUID(*CtlGuid*, \

(*aaaaaaaa,bbbb,cccc,dddd,eeeeeeeeeeee*), \

WPP\_DEFINE\_BIT(*TRACELEVELONE*) \

WPP\_DEFINE\_BIT(*TRACELEVELTWO*) )

The components of the definition have the following meanings:

* *CtlGuid* is a friendly name for the GUID.
* (*aaaaaaaa,bbbb,cccc,dddd,eeeeeeeeeeee*) represents the control GUID itself.

The letters represent the five fields of the standard string form of a GUID, but they are separated by commas and bracketed by parentheses instead of the usual hyphens and curly braces.

* *TRACELEVELONE* and *TRACELEVELTWO* are the provider’s two trace flags.

WPP assigns a bit value to each trace flag in the order in which they appear in the definition, beginning with 1.

### UMDF Example: Defining the Control GUID and Trace Flags

The Fx2\_Driver sample defines a single control GUID and four trace flags in the Internal.h header file, as shown in the following example:

#define WPP\_CONTROL\_GUIDS \

WPP\_DEFINE\_CONTROL\_GUID(WudfOsrUsbFx2TraceGuid, \

(da5fbdfd,1eae,4ecf,b426,a3818f325ddb), \

WPP\_DEFINE\_BIT(MYDRIVER\_ALL\_INFO) \

WPP\_DEFINE\_BIT(TEST\_TRACE\_DRIVER) \

WPP\_DEFINE\_BIT(TEST\_TRACE\_DEVICE) \

WPP\_DEFINE\_BIT(TEST\_TRACE\_QUEUE) \

)

In this example:

* WudfOsrUsbFx2TraceGuid is the friendly name for the {da5fbdfd-1eae-4ecf-b426-a3818f325ddb} GUID.
* The WPP\_DEFINE\_BIT macros specify four flags for the TraceEvents function’s FLAGS parameter.

These flags are used to distinguish between messages that are generated by the driver callback object, the device callback object, the queue callback objects, and all other driver components.

### KMDF Example: Defining the Control GUID and Trace Flags

The Osrusbfx2 driver sample defines a single control GUID and seven trace flags in the Trace.h header file, as shown in the following example:

#define WPP\_CONTROL\_GUIDS \

WPP\_DEFINE\_CONTROL\_GUID(OsrUsbFxTraceGuid, \

(d23a0c5a,d307,4f0e,ae8e,E2A355AD5DAB), \

WPP\_DEFINE\_BIT(DBG\_INIT) /\* bit 0 = 0x00000001 \*/ \

WPP\_DEFINE\_BIT(DBG\_PNP) /\* bit 1 = 0x00000002 \*/ \

WPP\_DEFINE\_BIT(DBG\_POWER) /\* bit 2 = 0x00000004 \*/ \

WPP\_DEFINE\_BIT(DBG\_WMI) /\* bit 3 = 0x00000008 \*/ \

WPP\_DEFINE\_BIT(DBG\_CREATE\_CLOSE) /\* bit 4 = 0x00000010 \*/ \

WPP\_DEFINE\_BIT(DBG\_IOCTL) /\* bit 5 = 0x00000020 \*/ \

WPP\_DEFINE\_BIT(DBG\_WRITE) /\* bit 6 = 0x00000040 \*/ \

WPP\_DEFINE\_BIT(DBG\_READ) /\* bit 7 = 0x00000080 \*/ \

)

In this example:

* OsrUsbFxTraceGuid is the friendly name for the {d23a0c5a-d307-4f0e-ae8e-E2A355AD5DAB} GUID.
* The trace flags are used to differentiate between trace messages that are generated as the driver handles different types of I/O requests.

## Initialize and Clean Up Tracing

You must initialize tracing early in the driver-loading process and clean up just before the driver shuts down.

### Initialize Tracing

The WPP\_INIT\_TRACING macro initializes tracing and must be called early in the loading process, before any trace messages are generated. WPP\_INIT\_TRACING is defined differently for user-mode and kernel-mode drivers and takes different arguments.

### UMDF Example: Initialize Tracing in DllMain

A UMDF driver should initialize tracing in its **DllMain** function by calling WPP\_INIT\_TRACING when the function is first called. In that case, the *Reason* parameter to **DllMain** is set to DLL\_PROCESS\_ATTACH.

**Important:** **DllMain** is also called when threads are attached and detached and when the process is detached. Be sure to verify that the *Reason* parameter is set to DLL\_PROCESS\_ATTACH before you enable tracing. Otherwise, you might attempt to enable tracing every time a thread is attached or detached.

For user-mode components, WPP\_INIT\_TRACING has a single argument: a pointer to a UNICODE string that identifies the driver. On Windows Vista and later versions, WPP ignores the string, and you can set it to NULL.

In the Fx2\_Driver sample, the string constant MYDRIVER\_TRACING\_ID is defined in Internal.h as shown in the following example:

#define MYDRIVER\_TRACING\_ID L"Microsoft\\UMDF\\OsrUsb"

The constant makes it possible to copy the Dllsup.cpp file and use it without change in another driver. However, you should redefine MYDRIVER\_TRACING\_ID to use an appropriate string for your driver.

The following example shows an edited version of the Fx2\_Driver sample’s **DllMain** implementation, from Dllsup.cpp:

BOOL WINAPI DllMain(HINSTANCE ModuleHandle,

DWORD Reason,

PVOID /\* Reserved \*/)

{

UNREFERENCED\_PARAMETER(ModuleHandle);

if (DLL\_PROCESS\_ATTACH == Reason)

{

WPP\_INIT\_TRACING(MYDRIVER\_TRACING\_ID);

}

...//Code omitted

return TRUE;

}

### KMDF Example: Initializing Tracing in DriverEntry

A KMDF driver should initialize tracing in its **DriverEntry** routine. For kernel-mode drivers, WPP\_INIT\_TRACING takes two arguments: pointers to the WDM driver object and to the registry path, both of which are passed as parameters to **DriverEntry**. On Windows Vista and later versions, WPP ignores these parameters, so you can set them to NULL.

**Note:** The procedures for initializing and cleaning up WPP tracing are different for Windows 2000 than for later versions of Windows. KMDF handles these differences internally, so you can use the following examples with any version of Windows that supports KMDF.

The following example shows how the Osrusbfx2 sample initializes tracing in its **DriverEntry** routine, from Device.c:

NTSTATUS DriverEntry(\_\_in PDRIVER\_OBJECT DriverObject,

\_\_in PUNICODE\_STRING RegistryPath)

{

WDF\_DRIVER\_CONFIG config;

NTSTATUS status;

WDF\_OBJECT\_ATTRIBUTES attributes;

WPP\_INIT\_TRACING( DriverObject, RegistryPath );

TraceEvents(TRACE\_LEVEL\_INFORMATION, DBG\_INIT,

"OSRUSBFX2 Driver Sample - Driver Framework Edition.\n");

...//Code omitted

}

### Clean Up Tracing

Drivers must call the WPP\_CLEANUP macro to clean up software tracing. A driver cannot generate trace messages after this macro is called, so it should be called immediately before the driver unloads.

Failing to call WPP\_CLEANUP leaves an outstanding reference on the driver object that can compromise subsequent attempts to load the driver. An outstanding reference can require users to reboot the system or it can generate a bug check when ETW attempts to reference the driver. The Build utility does not generate a compiler error if you do not call WPP\_CLEANUP.

### UMDF Example: Clean Up Tracing in DllMain

The UMDF version of WPP\_CLEANUP has no arguments. The macro is usually called in a UMDF driver's **DllMain** function, which is called when the driver is unloaded as well as when it is loaded. When the driver is being unloaded, the *Reason* parameter to **DllMain** is set to DLL\_PROCESS\_DETACH. The following example shows an edited version of the Fx2\_Driver sample’s **DllMain** implementation from Dllsup.cpp:

BOOL WINAPI DllMain(HINSTANCE ModuleHandle,

DWORD Reason,

PVOID /\* Reserved \*/)

{

UNREFERENCED\_PARAMETER(ModuleHandle);

...//Code omitted

else if (DLL\_PROCESS\_DETACH == Reason)

{

WPP\_CLEANUP();

}

return TRUE;

}

### KMDF Example: Cleaning up Tracing in *EvtCleanupCallback*

A KMDF driver should register an *EvtCleanupCallback* function for the WDFDRIVER object in its **DriverEntry** routine and call WPP\_CLEANUP from that function. The framework invokes *EvtCleanupCallback* immediately before it deletes the driver object, and thus immediately before it unloads the driver.

Because the framework calls *EvtCleanupCallback* only if the framework driver object is successfully created, you should initialize WPP only after you have successfully created the object. That way, if the creation attempt fails, WPP is not initialized and no cleanup is required. If the creation attempt is successful, the framework is guaranteed to call *EvtCleanupCallback*, which runs the cleanup code.

The kernel-mode version of WPP\_CLEANUP requires one argument, a pointer to the WDM driver object. The following example shows the Osrusbfx2 sample's *EvtCleanupCallback* implementation from Device.c:

VOID OsrFxEvtDriverContextCleanup(IN WDFDRIVER Driver)

{

PAGED\_CODE ();

TraceEvents(TRACE\_LEVEL\_INFORMATION,

DBG\_INIT,

"<-- OsrFxEvtDriverUnload\n");

WPP\_CLEANUP( WdfDriverWdmGetDriverObject( Driver ));

}

*EvtCleanupCallback* receives a pointer to the driver’s WDF driver object. To get a pointer to the underlying WDM driver object, the sample passes the WDF driver object handle to **WdfDriverWdmGetDriverObject**, which returns a pointer to the WDM driver object.

Cleaning up tracing in *EvtCleanupCallback* is sufficient for most scenarios. However, if the driver fails to create a WDF driver object in its **DriverEntry** routine, the framework has no driver object to delete and does not invoke *EvtCleanupCallback*. If you initialize tracing before you attempt to create the framework driver object, the driver code that handles the failure to create a driver object must call WPP\_CLEANUP.

The following example shows how the Osrusbfx2 **DriverEntry** routine registers the *EvtCleanupCallback* and cleans up tracing if a driver object is not successfully created:

NTSTATUS

DriverEntry(

IN PDRIVER\_OBJECT DriverObject,

IN PUNICODE\_STRING RegistryPath

)

{

WDF\_DRIVER\_CONFIG config;

NTSTATUS status;

WDF\_OBJECT\_ATTRIBUTES attributes;

WPP\_INIT\_TRACING(DriverObject, RegistryPath );

...//Omitted code

attributes.EvtCleanupCallback = OsrFxEvtDriverContextCleanup;

status = WdfDriverCreate(DriverObject,

RegistryPath,

&attributes,

&config,

&driver );

if (!NT\_SUCCESS(status)) {

TraceEvents(TRACE\_LEVEL\_ERROR, DBG\_INIT,

"WdfDriverCreate failed with status 0x%X\n",

status);

WPP\_CLEANUP(DriverObject);

}

return status;

}

## Instrument the Driver Code

You can place trace message function calls anywhere in the driver code that you find them useful. Typical locations are at the beginning or ending of routines, or in error handling paths.

### UMDF Example: Adding Trace Message Calls to Driver Code

The following example from Dllsup.cpp shows how to use a custom trace function—Trace—to return information about an error condition:

if (IsEqualCLSID(ClassId, CLSID\_MyDriverCoClass) == false)

{

Trace(TRACE\_LEVEL\_ERROR,

L"ERROR: Called to create instance of unrecognized class" "(%!GUID!)",

&ClassId);

return CLASS\_E\_CLASSNOTAVAILABLE;

}

This Trace call generates a trace message only if the trace controller enables the TRACE\_LEVEL\_ERROR level.

### KMDF Example: Adding Trace Message Calls to Driver Code

The following example shows how the Osrusbfx2 driver uses its TraceEvents function in a portion of the code devoted to handling read requests:

if (Length > TEST\_BOARD\_TRANSFER\_BUFFER\_SIZE) {

TraceEvents(TRACE\_LEVEL\_ERROR,

DBG\_READ,

"Transfer exceeds %d\n",

TEST\_BOARD\_TRANSFER\_BUFFER\_SIZE);

status = STATUS\_INVALID\_PARAMETER;

}

The call to TraceEvents generates a trace message if the trace controller enables the TRACE\_LEVEL\_ERROR level and the DBG\_READ trace flag. The message includes the value of the driver-defined constant TEST\_BOARD\_TRANSFER\_BUFFER\_SIZE.

# Tools for Software Tracing

The WDK includes a set of tools for software tracing. These tools are designed to support drivers that use ETW tracing and to supplement the tracing tools in Windows. The tools are located under WinDDK\*BuildNumber*\tools\tracing\, with separate sets of tools for each of the three supported CPU architectures. The tools include the following:

Tracefmt

A command-line trace consumer that formats trace messages and writes them to files or displays them in the command window.

Tracelog

A command-line trace controller that supports user-mode and kernel-mode trace sessions.

Tracepdb

A command-line tool that creates TMF files from symbol files.

TraceView

A GUI-based trace controller and trace consumer that combines and extends the features of Tracepdb, Tracelog, and Tracefmt. This tool is designed for real-time displays of trace messages but can also be used to view trace log files.

TraceWPP

A command-line tool that runs the WPP preprocessor on a trace provider’s source files. This tool is an alternative to running the preprocessor as part of the regular build process.

Two additional tracing tools are included with Windows XP and later:

Logman

A fully functional, GUI-based trace controller designed to control the logging of performance counters and event traces.

Tracerpt

A command-line trace consumer that formats trace events and performance counters and writes them to files that can be read by tools, such as Microsoft Excel. Tracerpt also analyzes the events and generates summary reports.

The Control Panel Event Viewer application cannot be used to display trace logs generated by drivers that use WPP.

# How to Run a Software Trace Session

A software trace session requires the following three components:

* A trace provider to generate trace messages.
* A trace controller to enable tracing and create a trace log that can be displayed by the trace consumer.
* A trace consumer to format and display the trace log.

TraceView, which is used in some of the following examples, acts as both a controller and a consumer. You can also use separate applications for controller and consumer.

## Prepare the Driver

Before starting the trace session, you must build and install the driver and create TMF and control GUID (.ctl) files. The procedure for Fx2\_Driver is identical to the one for Osrusbfx2 except for some obvious name changes.

To prepare the driver

1. Build and install the sample on your test system.

2. Build and install the test application on your test system.

3. If you have not installed the WDK on your test system, copy TraceView from your development system to a convenient location on your test system.

4. Open a command window with elevated privileges, use **cd** to move to the project’s target folder, and run Tracepdb to generate TMF files from the sample’s symbol file. The following example shows the command for an x86-based computer:

\WinDDk\*BuildNumber*\tools\tracing\i386\tracepdb -f  
 *drivername*.pdb -p *path*

If you are running Tracepdb on a 64-bit system, use the version in the /x64 or /ia64 folder.

The resulting TMF files are named with a GUID and have a .tmf extension. The files are stored in the *path* folder. Typically, developers set *path* to “.” to store the TMF files in the project’s target folder.

**Note:** In Windows Server 2003 and earlier versions, you must copy the Dbghelp.dll file from WinDDK\*BuildNumber*\bin\Platform into the folder in which Tracepdb.exe is located.

5. Create a CTL file.

This is a text file with a single line of text that contains the control GUID and provider name from WPP\_DEFINE\_CONTROL\_GUID. The file can have any name you prefer, but it must have a .ctl extension.

For the Fx2\_Driver sample, the CTL file should contain the following line of text:

da5fbdfd-1eae-4ecf-b426-a3818f325ddb WudfOsrUsbFx2TraceGuid

For the Osrusbfx2 sample, the CTL file should contain the following line of text:

d23a0c5a-d307-4f0e-ae8e-E2A355AD5DAB OsrUsbFxTraceGuid

## View a Trace Log by Using TraceView

Trace logs can be viewed in real time or stored in a log file to be viewed later. This section describes how to use TraceView to create and view a trace log file and to view a trace log in real time.

### How to Create and View a Trace Log File

One way to handle trace messages is to store them in a trace log file. You can then use a trace consumer to view the file later. This section shows how to store the trace messages from Fx2\_Driver in a trace log file and then view them by using TraceView.

To start the trace session

1. Install Fx2\_Driver and TraceView on your test computer.

2. Run TraceView by double-clicking it in Windows Explorer or by using a command line.

In Windows Vista and later versions, you must run TraceView with elevated privileges.

3. On the **File** menu, click **Create New Log Session**, and then click **Add Provider**.

4. In the **Provider Control GUID Setup** dialog box, select **CTL (Control GUID) File**, enter the path to the sample’s CTL file, and then click **OK**.

5. In the **Format Information Source Select** dialog box, select **Select TMF Files**, and then click **OK**.

6. In the **Trace Format Information Setup** dialog box, click **Add**.

7. In the **File Open** dialog box, enter the names of the TMF files that you created earlier.

8. In the **Trace Format Information Setup** dialog box, click **Done**.

9. In the **Create New Log Session** dialog box, click **Next**.

10. On the **Log Session Options** page, select **Log Trace Event Data To File**, specify a name for the log file, and then click **Finish** to complete setting up the trace session.

The Fx2\_Driver sample has trace messages only in its load and startup code. To generate a trace message log, unplug the device and plug it back in.

To view the messages in the log file

1. On the TraceView **File** menu, click **Open Existing Log File**.

2. In the **Log File Selection** dialog box, enter the name of the trace log file that you created in the previous procedure, and then click **OK**.

3. In the **Format Information Setup** dialog box, specify the names of the project’s TMF files, and then click **Done** to view the trace message log.

Figure 2 shows a trace log file for Fx2\_Driver, produced by unplugging and replugging the device while the trace session is running.

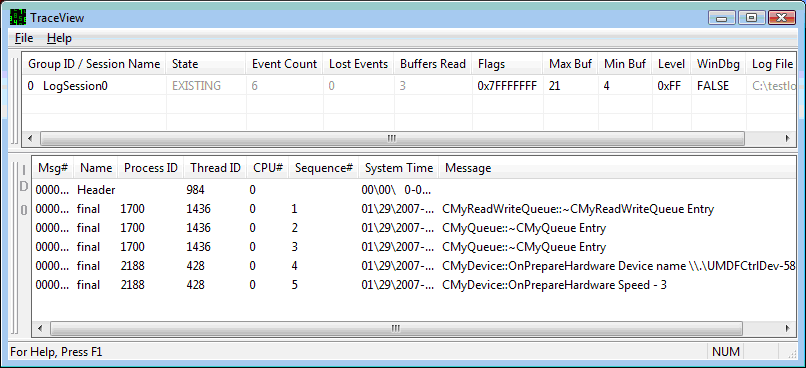


Figure 2. Viewing a trace log file by using TraceView

### How to View a Trace Log in Real Time

Viewing a trace log in real time allows you to immediately see how the driver responds to different events. This section describes a simple real-time trace session with the Osrusbfx2 sample.

To start the trace session

1. Install the Osrusbfx2 driver, the test application, and TraceView on your test computer.

2. Run TraceView.

In Windows Vista and later versions, you must run TraceView with elevated privileges.

3. On the **File** menu, click **Create** **New** **Log** **Session**, and then click **Add Provider**.

4. In the **Provider Control GUID Setup** dialog box, select **CTL (Control GUID) File**, enter the sample’s CTL file path, and then click **OK**.

5. In the **Format Information Source Select** dialog box, select the **Select TMF Files** option, and then click **OK**.

6. In the **Trace** **Format Information Setup** dialog box, click **Add**.

7. In the **File Open** dialog box, enter the names of the TMF files that you created earlier, and then click **Done**.

8. In the **Create New Log Session** dialog box, click **Next**.

9. On the **Log Session Options** page, select **Real Time Display**, and then click **Finish** to complete setting up the trace session.

You can also direct TraceView to store the messages a trace log file in addition to displaying them in real time.

To generate trace messages, run the test application and change one of the device features. Figure 3 shows the trace messages that are produced by lighting two lights on the light bar of the Fx2 device:

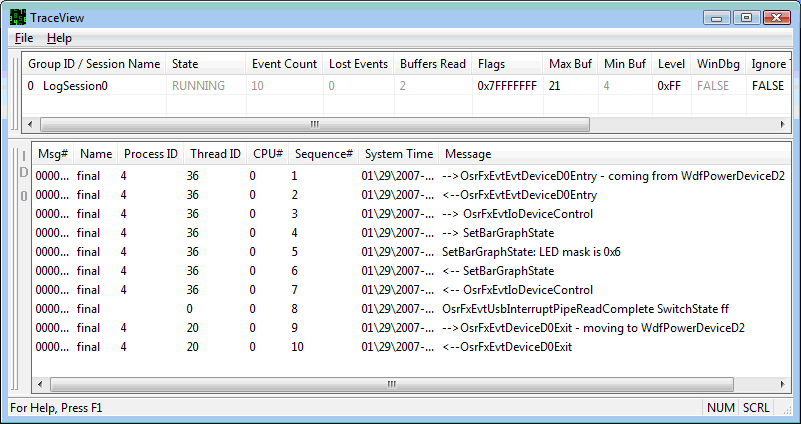


Figure 3. A real-time TraceView session for Osrusbfx2

## View Trace Messages in Real Time with WinDbg

You can configure WinDbg to receive and display trace messages from a WDF driver, so that you can view the messages in the WinDbg **Command** window in real time as you debug the driver. This example shows how to use TraceView to redirect Osrusbfx2 trace messages to the WinDbg kernel debugger.

To configure WinDbg to display trace messages

1. Launch WinDbg, put it in kernel-debugging mode, and connect to the target computer.

2. Use the **.load** command to load Wmitrace.dll and Traceprt.dll.

These files contain tracing-related debugger extensions. They are located in the Program Files\Debugging Tools for Windows\Winxp folder.

3. Copy the driver’s TMF files to a convenient folder on the host computer.

4. Use the **!wmitrace.searchpath** debugger extension to specify the location of the TMF files.

To format the trace messages, the debugger must be able to find the TMF files.

To start a debugging session that displays trace messages

1. Run TraceView on the test computer and create a new trace log session.

2. In the TraceView **Log Session Options** page, select **Real Time Display**, and then click **Advanced Log Session Options**.

3. In the **Advanced** **Log Session Options** page, on the **Log Session Parameter Options** tab, change the value of the **WinDbg** option to **TRUE**, and then click **OK**.

4. Click **Finish** to complete the process of creating the trace log session.

5. Run the Osrusbfx2 test application and make some changes in the display to generate trace messages.

Figure 4 shows how WinDbg displays trace messages that were generated by changing the lights on the device’s light bar.

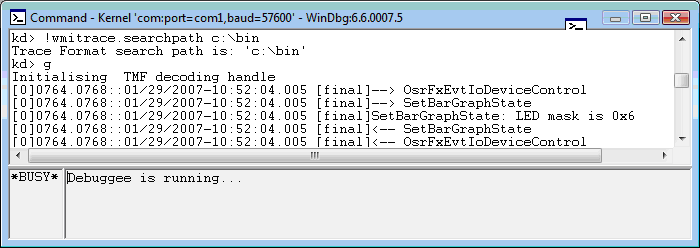


Figure 4. Viewing trace messages with WinDbg

## View the Frameworks Trace Log by Using the Core Tracing Tools

The Tracelog and Tracefmt core tracing tools are less convenient than TraceView. However, they are more robust, which makes them the preferred tools in some situations. This section shows how to use the core tracing tools to view trace messages from the KMDF runtime.

To use Tracelog to view messages from the KMDF runtime, you must set several environment variables and then start a Tracelog session. On Windows Vista systems, you must open the command window and run Tracelog with elevated privileges.

To set the environment variables, open a command window and run the following commands:

set TRACE\_NAME=**WDF**

set TRACE\_LOG\_PATH=**%SystemRoot%\Tracing\%TRACE\_NAME%**

set \_PROVIDER\_GUID=**#544d4c9d-942c-46d5-bf50-df5cd9524a50**

set TMFFILE=**wdf01005.tmf**

set TRACE\_FORMAT\_PREFIX=%%2!-20.20s!%%!FUNC!-

set TRACE\_FLAGS=**0xffff**

set TRACE\_LEVEL=**5**

md "%TRACE\_LOG\_PATH%"

These commands set environment variables and create a folder for the trace log file. You should modify the highlighted parts of the commands—especially the GUID and the TMF file name—to use values appropriate for your driver.

To start a Tracelog session, run the following command:

tracelog -start "%TRACE\_NAME%" -seq 10 -rt -guid %\_PROVIDER\_GUID% -flags %TRACE\_FLAGS% -level %TRACE\_LEVEL% -f "%TRACE\_LOG\_PATH%\%TRACE\_NAME%.etl"

By default, the messages are placed in a trace log file named %TRACE\_NAME% that is placed in the %TRACE\_LOG\_PATH% folder.

You can also view the messages in real time. To do so, after you have started a Tracelog session, run the Tracefmt tool as follows:

tracefmt -rt "%TRACE\_NAME%" -tmf "%TMFFILE%" -display

Tracefmt displays the messages in the command window as they are generated, using the formatting information in the specified TMF file. The following example shows how Tracelog and Tracefmt display some of the framework trace messages that were generated by lighting two of the lights on the Fx2 device’s light bar:

C:\WinDDK\6000\tools\tracing\i386>tracefmt.exe -rt "%TRACE\_NAME%" -tmf

"%TMFFILE%" -display

Setting RealTime mode for WDF

Examining wdf01005.tmf for message formats, 97 found.

Searching for TMF files on path: (null)

FxPkgGeneral\_cpp424 FxPkgGeneral::Dispatch- WDFDEVICE 0x6C3BC458

!devobj 0x93C44810 0x00000000(IRP\_MJ\_CREATE) IRP 0x9061A008

FxPkgIo\_cpp97 FxPkgIo::Dispatch- WDFDEVICE 0x6C3BC458

!devobj 0x93C44810 0x0000000e(IRP\_MJ\_DEVICE\_CONTROL), IRP\_MN 0,

IRP 0x9331FED8

FxDevice\_cpp1646 FxDevice::AllocateRequestMemory- Allocating

FxRequest\* 9048E308, WDFREQUEST 6FB71CF0

FxIoQueue\_cpp1696 FxIoQueue::QueueRequest- Queuing WDFREQUEST

0x6FB71CF0 on WDFQUEUE 0x6C3BAFB0

FxIoQueue\_cpp2085 FxIoQueue::DispatchEvents- Thread 9E867AC0

is processing WDFQUEUE 0x6C3BAFB0

PowerPolicyStateMachFxPkgPnp::PowerPolicyCancelWaitWake-

Successfully got WaitWake irp 93C49750 for cancelling

PowerPolicyStateMachFxPkgPnp::\_PowerPolicyWaitWakeCompletionRoutine-

Completion of WaitWake irp 93C49750, 0xc0000120(STATUS\_CANCELLED)

PowerPolicyStateMachFxPkgPnp::\_PowerPolicyWaitWakeCompletionRoutine-

Not completing WaitWake irp 93C49750 in completion routine

PowerPolicyStateMachFxPkgPnp::PowerPolicyCancelWaitWake- Cancel of

irp 93C49750 returned 1

FxPkgPnp\_cpp494 FxPkgPnp::Dispatch- WDFDEVICE 0x6C3BC458

!devobj 0x93C44810 IRP\_MJ\_POWER, Minor 0x2 IRP 0x93C49750

PowerPolicyStateMachFxPkgPnp::PowerPolicySendDevicePowerRequest-

Requesting D0 irp, 0x00000103(STATUS\_PENDING)

FxIoTarget\_cpp158 FxIoTarget::SubmitPendedRequest- Sending

WDFREQUEST 6CE5CCC8, Irp 931A52F8

FxIoTarget\_cpp158 FxIoTarget::SubmitPendedRequest- Sending

WDFREQUEST 6C3BB268, Irp 93C49B38

FxIoQueue\_cpp2085 FxIoQueue::DispatchEvents- Thread 820FBD78

is processing WDFQUEUE 0x6C3BAFB0

FxIoQueue\_cpp2170 FxIoQueue::DispatchEvents- WDFQUEUE

0x6C3BAFB0 Power Transition State 0x0000000a(FxIoQueuePowerRestarting)

You can flush the messages from the trace buffer at any time during a trace session without stopping the session.

To flush the trace buffer and view the messages without stopping the session

1. Run the following command to flush the buffer:

tracelog -flush "%TRACE\_NAME%"

2. Then run this command:

tracefmt "%TRACE\_LOG\_PATH%\%TRACE\_NAME%.etl" -tmf %TMFFILE% -nosummary -o "%TRACE\_LOG\_PATH%\%COMPUTERNAME%-%TRACE\_NAME%.txt"

3. Finally, run this command to dump the messages in a file:

@echo Tracelog dumped to %TRACE\_LOG\_PATH%\%COMPUTERNAME%-%TRACE\_NAME%.txt

To stop the trace session, run the following command:

tracelog -stop "%TRACE\_NAME%"

# Best Practices: Design for Diagnosability

Generating trace messages is a relatively easy way to record what your driver is doing. You can use the messages to track the path of an I/O request through your code, to figure out where bugs might be, and determine which routines are called frequently and which are rarely exercised. With this information, you can diagnose driver problems more easily.

To get the most out of tracing, observe these best practices:

* Define trace flags that correspond to the types of information that you find most helpful during debugging.

For example, as you incrementally implement and test your code, you could define a new flag for each new driver module.

* Do not conditionally compile the trace statements in your driver.

Leave them in the retail binaries—trace messages have a noticeable performance impact only if a trace consumer explicitly enables them.

* Place a trace message in every error path.

The message can provide detailed information to help diagnose the cause of the error with relatively little overhead.

# Resources

The following resources provide additional information on software tracing in general and WPP tracing in particular.

Controlling Event Tracing Sessions

<http://go.microsoft.com/fwlink/?LinkId=80062>

Adding Event Tracing to Kernel-Mode Drivers

<http://go.microsoft.com/fwlink/?LinkId=80610>

DoTraceMessage

<http://go.microsoft.com/fwlink/?LinkId=80611>

WPP Software Tracing

<http://go.microsoft.com/fwlink/?LinkId=80090>

Can I customize DoTraceMessage

<http://go.microsoft.com/fwlink/?LinkId=80061>

Using UUIDGEN.EXE

<http://go.microsoft.com/fwlink/?LinkId=79587>