Microsoft Windows

Getting Started with Software Tracing in Windows Drivers

WinHEC 2005 Version - April 18, 2005

Abstract

Software tracing is an efficient way to examining the behavior of a running driver without the overhead of debug print statements or a checked build. This paper provides a general introduction to software tracing in kernel-mode drivers for the Microsoft® Windows® family of operating systems using Event Tracing for Windows (ETW) and the Windows software trace preprocessor (WPP). It also describes advances in software tracing for Microsoft Windows Vista™.

This information applies for the following operating systems:

- Windows Vista
- Microsoft Windows Server™ 2003
- Microsoft Windows XP
- Microsoft Windows 2000

Future versions of this preview information will be provided in the Windows Driver Kit.

The current version of this paper is maintained on the Web at http://www.microsoft.com/whdc/DevTools/tools/WPP_Intro.mspx.

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

Contents

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

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

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

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.

Microsoft, Windows, 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.

Introduction

Software tracing is an efficient way of examining the behavior of a running driver, without the overhead of debug print statements or a checked build. Drivers that are instrumented for tracing can generate trace messages for events determined by the driver writer. Areas that can be useful to trace include:

- Configuration settings.
- State changes, to record the cause of the state change and any related information.
- Input and results of public functions.
- Scavenging or cleanup operations.
- Memory allocations.
- I/O activities such as I/O request packet (IRP) completion, IRP cancellation, and so forth.
- Error paths, especially in libraries that other developers will use. Placing a detailed trace statement in every error path can generate useful information for a library's users with minimal overhead.

Microsoft® Windows® supports software tracing through Event Tracing for Windows (ETW), a kernel-level trace logging facility that logs both kernel and application-defined events. The Windows software trace preprocessor (WPP) simplifies and enhances ETW tracing in drivers. Drivers for Microsoft Windows 2000 and later versions of Windows can take advantage of ETW and WPP to support software tracing.

This paper provides a general introduction to software tracing in Windows kernel-mode drivers. For detailed information about implementing software tracing in drivers and using tracing tools, see the Windows Driver Kit (WDK).

For information about instrumenting a user-mode application for software tracing, see the Platform SDK.

Advantages of Software Tracing for Drivers

Software tracing provides several significant advantages for drivers:

- **Available in shipped products.** Because tracing does not require the checked build of Windows, product binaries can be shipped with tracing code left in. Tracing can be enabled and a trace log captured on a single system in the field by using the tracing tools included with Windows. Use of a kernel-mode debugger is not required.
- **Low impact on performance.** Drivers generate trace messages only when tracing is enabled—otherwise, a driver's tracing code is never called. When tracing is enabled, it incurs little overhead because trace messages are issued in binary format and formatted outside the driver by a separate application. This makes tracing especially useful for investigating bugs whose behavior is affected by system performance (sometimes called "Heisenbugs" because they tend to disappear when something slows down the system, as debug print function calls tend to do).
- **Dynamic and flexible.** Tracing can be enabled or disabled dynamically, without stopping or restarting a component and without rebooting the operating system. Tracing can be enabled selectively for individual components, and it can be enabled at different levels defined by the driver writer.

In Microsoft Windows XP and later versions of Windows, trace messages can be redirected to a real-time trace consumer (any program capable of receiving and formatting trace messages) or a kernel debugger, bypassing the trace log.

- **Rich trace information without exposing intellectual property.** Developers often put function names and other internal information in debug print statements. Although this is helpful during debugging, it also simplifies reverse engineering, which can expose your company's intellectual property.

  With ETW, all of the static portions of a trace message are placed in the driver's private symbol file. The function name, source file name, and line number for each trace message are added during the preprocessing phase of trace implementation. When the message is logged, the ETW logging mechanism automatically attaches a timestamp to each trace message (plus any other dynamic information specified by the driver).

  The logged trace messages can be read only by a trace consumer (such as the Tracefmt tool provided with the WDK) that has access to either the driver's program database (PDB) symbol file or to a trace message format (TMF) file generated from the PDB file.
- **Easy migration from debug print statements.** The WPP can interpret a driver's existing debug print function calls as trace function calls during the build process, so a driver's existing instrumentation can be used and extended without having to rewrite existing code.

Software Tracing Concepts

Figure 1 shows software tracing components and how they interact.

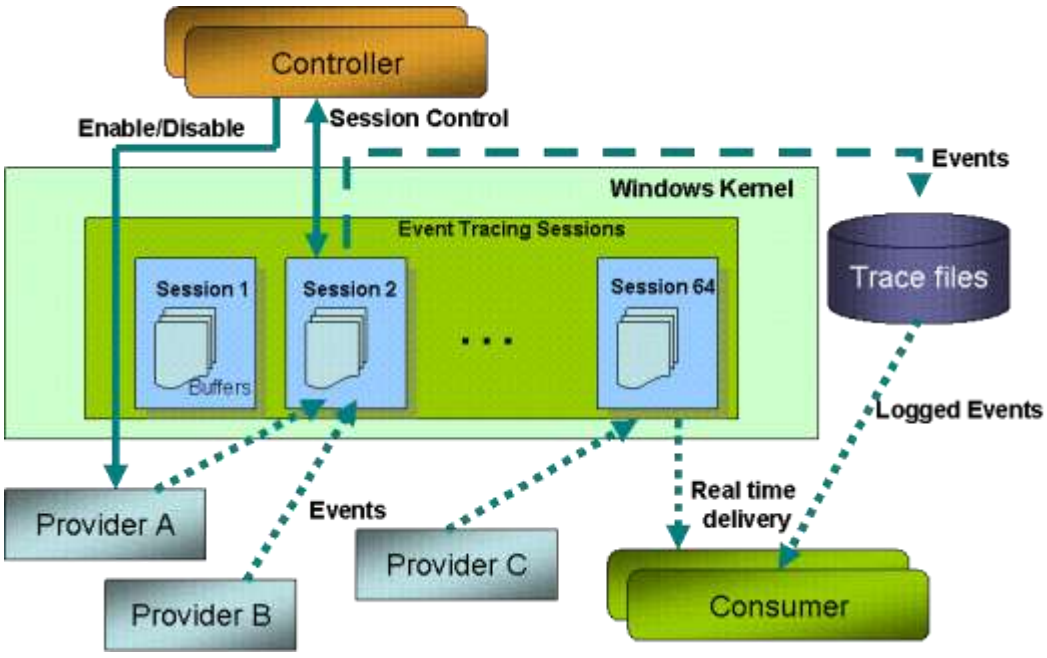

Figure 1. Software Tracing Architecture

Trace Providers

A *trace provider* is an application, operating system component, or driver that is instrumented for tracing. Figure 1 shows three trace providers: Provider A, Provider B, and Provider C (bottom of figure). "Instrumented for tracing" means that the trace provider includes code to generate trace messages. A number of

Windows kernel-mode components, drivers, and Microsoft Windows Server™ applications are instrumented as trace providers. Developers can instrument their drivers by using the WPP or ETW, as described in this paper.

A trace provider generates trace messages only when it has been enabled by a *trace controller*, which is an application or tool that manages trace sessions. TraceView and TraceLog are trace controllers provided with the WDK. (You can also write your own trace controller by using the ETW application programming interface documented in the Platform SDK.)

Trace Sessions

A *trace session* is a period during which one or more trace providers generate trace messages to a single event log. The trace controller starts and configures a trace 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.

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 or trace consumer once per second, or sooner if the buffers become full. Buffers are also flushed automatically when the trace controller stops the trace session. In addition, a trace controller can explicitly flush the buffers, either on demand or at regular intervals. If the system crashes, the buffers are still available, so trace messages are not lost.

A single trace session can record events from more than one provider. Trace messages from all providers that are associated with a session are interleaved in the session's buffers. Figure 1 shows three trace sessions: Session 1, Session 2, and Session 64. Provider A and Provider B are associated with Session 2, whereas Provider C is associated with Session 64. A trace session can exist without having a provider associated with it. For the moment, no trace provider is associated with Session 1.

On Windows XP and later versions of Windows, ETW supports up to 64 trace sessions executing simultaneously. (On Windows 2000, ETW supports up to 32 sessions.) All but two of these sessions are available for general use. The remaining two sessions are the global logger trace session (which records events that occur early in the operating system boot process) and the NT kernel logger trace session (which records predefined system events generated by the operating system, such as disk I/O or page fault events). It is possible to trace both the Windows kernel and a driver and produce a single trace log with events from both components interleaved, thus capturing the actions of Windows while the driver is running.

Trace Controllers

The trace controller transfers trace messages from the trace session buffers either to an *event trace log* (ETL) *file* for that session or directly to a *trace consumer*, which is an application or tool that receives, formats, and displays trace messages. (A trace consumer can also just format and display trace messages from the trace log, long after the trace session has ended.)

The trace consumer formats trace messages in human-readable format by using instructions either from the provider's PDB symbol file or from a *TMF file* that is generated when the provider is built (or generated later from the provider's PDB file, by using a tool such as Tracepdb).

ETW and the WPP

The underlying architecture of Windows kernel-mode tracing is based on ETW. Though ETW tracing can be powerful and flexible, it is also complex to implement, especially for drivers that need only lightweight tracing.

To implement ETW tracing, a driver must provide schemas in a managed object format (MOF) file for its trace events, register as a WMI provider, mark events as traceable, specify a control event for overall enabling and disabling of a set of trace events, store a handle to the logger when it receives a request from WMI to enable events, and decide whether a given trace event should be sent to WMI event consumers or to the WMI event logger only. When an event is triggered, the driver must allocate and fill in an EVENT_TRACE_HEADER structure and then (finally) call **IoWMIWriteEvent** to log the event.

The WPP simplifies and enhances ETW tracing in drivers and requires considerably less effort on the part of the developer. The WPP macros in driver source code and the WPP template files define C/C++ tracing code to create. During the build process, the WPP uses these macros and templates to generate code for the driver to register as a trace provider and to generate trace messages. The code generated for each source code file is stored in a *trace message header* (TMH) *file*, which also contains macros that add trace message formatting instructions to the driver's PDB file. The tracing code that the WPP generates essentially implements ETW tracing on behalf of the driver.

Instrumenting a Driver for Software Tracing Using the WPP

To instrument a driver for software tracing using the WPP, you add certain preprocessor directives and the WPP macros to your driver source code. You also add trace message calls to driver source code (or instruct the WPP how to convert existing debug print statements) for the messages you want to generate.

The *tracedrv* sample in the WDK consists of a simple driver called *tracedrv* and a test program called *tracectl. Tracedrv* implements software tracing through the WPP. *Tracectl* loads and starts *tracedrv* and sends an I/O control (IOCTL) to the driver to trigger trace messages. When *tracedrv* receives the IOCTL, it logs 100 trace messages and then stops. Source code for *tracedrv* and *tracectl* is in the %winddk%\src\general\tracedrv directory.

This section describes the basic steps for instrumenting a driver for software tracing using the WPP, illustrated with examples from *tracedrv*. These steps are:

1. Include the TMH file for each source code file that contains any WPP macros.
2. Define a WPP_CONTROL_GUIDS macro that specifies a globally unique identifier (GUID) and trace flag names for controlling tracing.
3. Call WPP_INIT_TRACING from the driver's **DriverEntry** routine.
4. Call WPP_CLEANUP from the driver's *Unload* routine.
5. Add a RUN_WPP statement to the driver's sources file.
6. Add trace message function calls to driver code.

Step 1. Include the TMH file for each source code file that contains any WPP macros.
For each source file that contains any WPP macros, add an **#include** directive for the corresponding TMH file. The WPP generates TMH files during the build process.

A TMH file has the same name as the corresponding source file, with a .tmh filename extension. For example, if the source code file is tracedrv.c, the corresponding TMH file would be tracedrv.tmh and the **#include** directive would look like the following:

```
#include tracedrv.tmh
```

Include the trace message header file before any WPP macro calls and after defining a WPP_CONTROL_GUIDS macro.

Step 2. Define a WPP_CONTROL_GUIDS macro that specifies a GUID and trace flag names for controlling tracing.
The WPP_CONTROL_GUIDS macro establishes a control GUID for the trace provider and defines trace flag names. ETW and the WPP both use the control GUID to identify the provider. Trace flags can be enabled by a trace controller (for example, with the **-flag** option used with the Tracelog tool) to determine which messages the trace provider generates. The provider interprets the flag as a condition for generating the message. (Trace levels defined in *evntrace.h* can also be used as conditions for generating trace messages, as described in "Step 6. Add trace message calls to driver code" later in this paper.)

Use a tool such as GUIDGEN to generate a GUID to serve as the control GUID. Define trace flags to distinguish between different levels or categories of tracing in your driver. For example, you might define certain trace flags for error paths and other trace flags for performance.

The tracedrv sample includes the following definition of the WPP_CONTROL_GUIDs macro:

```
#define WPP_CONTROL_GUIDS \
    WPP_DEFINE_CONTROL_GUID(CtlGuid,(d58c126f, b309, 11d1,
969e, 0000f875a5bc),  \
        WPP_DEFINE_BIT(TRACELEVELONE)                \
        WPP_DEFINE_BIT(TRACELEVELTWO) )
```

In this example, CtlGuid is the friendly name for the GUID {d58c126f-b309-11d1-969e-0000f875a5bc} and the trace flag names are TRACELEVELONE and TRACELEVELTWO. The WPP assigns bit values to a provider's trace flags in the order in which they appear in the WPP_CONTROL_GUIDS structure, beginning with 1. When starting a trace session, use the bit value (not the flag name) to represent the flag.

The control GUID and its friendly name are also specified in the driver's *control GUID file*, a text file that is used by the Tracelog tool. For example, *tracedrv.ctl*, the control GUID file for the *tracedrv* sample, contains the following line:

```
d58c126f-b309-11d1-969e-0000f875a5bc    CtlGuid
```

A driver can specify more than one control GUID. Each control GUID identifies a unique provider. For example, if a driver defines two control GUIDs, one for a shared library and one for the driver, the library and the driver can be enabled as two different providers. Tracing can be enabled for either the library or the driver, or both.

In contrast, all components that use the same control GUID operate as a single provider. For example, if several drivers use the same control GUID, they can be enabled as a single trace provider. These drivers share the ETW structures and resources, so their trace messages are interleaved in the buffers. This can be useful for tracing drivers that interact during execution, such as a kernel-mode port driver and its related miniport driver. Components can be user-mode or kernel-mode components, such as a kernel-mode driver and its associated user-mode control or application programs. This enables a provider to produce a consistent trace between user mode and kernel mode.

Step 3. Call WPP_INIT_TRACING from the driver's DriverEntry routine.
The WPP_INIT_TRACING macro activates software tracing in a driver. The driver cannot generate trace messages until after this macro has been called, so it should be called as early as possible.

For Windows XP and later versions of Windows. Call this macro from the driver's **DriverEntry** routine, using the following format:

```
WPP_INIT_TRACING(DriverObject,RegistryPath);
```

where *RegistryPath* is a pointer to a Unicode string that specifies the path to the driver's registry key.

For Windows 2000. If the driver must load on Windows 2000 and does not use WMI, call the WPP_SYSTEMCONTROL(*DriverObject*) macro from the driver's **DriverEntry** routine, to set up the WPP dispatch routine that handles WMI IRPs. If the driver uses WMI, see "Resources" at the end of this paper for an alternative solution.

Then call the WPP_INIT_TRACING macro from the driver's *AddDevice* routine and use the following format. Notice that the WPP_INIT_TRACING macro takes a device object (not a driver object) as its first argument:

```
WPP_INIT_TRACING(DeviceObject, RegistryPath)
```

Step 4. Call WPP_CLEANUP from the driver's *Unload* routine.
The WPP_CLEANUP macro deactivates software tracing in a driver. The driver cannot generate trace messages after this macro has been called, so it should be called as late as possible.

Be aware that failing to call this macro does not generate a compiler error, but it does leave an outstanding reference on the driver object, which causes subsequent loads of the driver to fail and requires the user to reboot the system.

For Windows XP and later versions of Windows. Call this macro from the driver's *Unload* routine, using the following format:

```
WPP_CLEANUP(DriverObject)
```

For Windows 2000. Call this macro just before calling *DeleteDevice* on the driver's device object, using the following format. Again, notice that the WPP_CLEANUP macro takes a device object (not a driver object) as its argument:

```
WPP_CLEANUP(DeviceObject)
```

Step 5. Add a RUN_WPP directive to the driver's sources file.
The WPP is integrated with the WDK build environment. The RUN_WPP directive invokes the WPP before compiling the driver files.

The WPP processes WPP macros and creates a TMH file for each source file. A TMH file is a text file that contains declarations of functions and variables used by the tracing code that the WPP generates. The TMH file also includes macros that add trace message formatting instructions to a driver's PDB file.

The RUN_WPP directive should be added to the end of the driver's *sources* file. In its simplest form, the RUN_WPP directive for a kernel-mode driver looks like the following:

```
RUN_WPP= $(sources) -km
```

The **-km** option defines the WPP_KERNEL_MODE macro, which traces kernel-mode components (by default, only user-mode components are traced). This directive is sufficient for drivers that run on Windows XP and later versions of Windows.

For details about **-km** and other RUN_WPP options, see the WPP documentation in the WDK.

As an alternative to running the WPP when building your driver, you can use the TraceWPP command-line tool to run the WPP on a source file that includes the WPP macros. TraceWPP uses the same parameters as RUN_WPP. This tool is provided with Windows and documented in the Platform SDK.

For Windows 2000. To support providers that run on Windows 2000 as well as later versions of Windows, build your driver with the template *km-w2k.tpl* (for a kernel-mode provider) or *um-w2k.tpl* (for a user-mode provider).

These templates support cross-platform compatibility by adding support for functions that might otherwise be found only in later versions of Windows, such as high-resolution timers and sequence numbers. These templates make it unnecessary to build a separate version of the driver for Windows 2000. These templates are provided with the WDK.

To use the templates, add the **-gen** parameter to the RUN_WPP macro in the provider's *sources* file, as shown in the following example for a kernel-mode provider:

```
RUN_WPP=$(SOURCES) -km -gen:{km-w2k.tpl}*.tmh
```

The **-gen** parameter directs the WPP to create a trace message header file for the *km-w2k.tpl* template. The resulting trace message header file is larger than it would be on Windows XP and later versions of Windows, so software tracing on Windows 2000 might have a slightly higher performance overhead.

Step 6. Add trace message calls to driver code.
A trace message function is the code that actually generates a trace message if the function's conditions are met. ETW supports trace flags and trace levels as conditions that determine which events the trace provider generates.

The default WPP trace macro, **DoTraceMessage**, generates trace messages on the basis of trace flags. The trace provider defines trace flags in its WPP_CONTROL_GUIDS structure, as described earlier in this paper. If a trace controller enables the trace flag, the provider generates the message.

The trace level typically represents the severity of an event (information, warning, or error), but a trace provider can interpret trace levels to represent any condition for generating the trace message. Trace levels are defined in the public header file *evntrace.h*.

You can define your own trace message functions that generate messages on the basis of any condition that is supported by your code, including a combination of trace flags and trace levels. You can also convert existing debug print statements to trace messages when you build your driver.

Important. Be careful not to expose security or privacy data in trace messages. Any trace consumer with access to your driver's TMH file or PDB file can format and read your driver's trace log.

The DoTraceMessage Macro

DoTraceMessage generates a trace message if a specified trace flag is enabled. A trace controller enables one or more flags when it starts a session.

The **DoTraceMessage** macro has the following format:

```
DoTraceMessage(Flag, "Message", MessageVariables...);
```

In this macro:

Flag is the name of a trace flag specified in the driver's definition of the WPP_CONTROL_GUIDS macro. The flag is a condition for generating the trace message specified in *Message*.

Message specifies a string constant that defines the text of the trace message, including format specifications (%d, %s, and the like). The string constant in *Message* must be compatible with the **FormatMessage** function.

MessageVariables specifies a comma-separated list of driver-defined variables whose formatted values are added to the message. The format of this list is the same as for **printf**.

FormatMessage and **printf** are both documented in the Platform SDK.

When the WPP encounters a **DoTraceMessage** call, it treats the message information as a unique message and does the following:

- Defines macros in the driver's TMH file that ultimately expand to **WmiTraceMessage** calls.
- Creates a compiler directive that adds tracing information to the driver's PDB file, including the required control GUID and information to format the provider's trace messages. Trace consumers use this information to identify the trace provider and format its trace messages.

For example, the *tracedrv* sample logs two different messages, each controlled by the trace flag TRACELEVELONE. Assuming TRACELEVELONE is enabled, each time *tracedrv* receives an IOCTL_TRACEKMP_TRACE_EVENT request, it generates the first trace message once and then generates the second message repeatedly, up to the value of MAXEVENTS as defined in *tracedrv.c*.

```
case IOCTL_TRACEKMP_TRACE_EVENT:
    ioctlCount++;
    DoTraceMessage(TRACELEVELONE, "IOCTL = %d", ioctlCount);

    while (i++ < MAXEVENTS) {
        DoTraceMessage(TRACELEVELONE,  "Hello, %d %s", i, "Hi" );
    }
}
```

In addition to the standard format strings defined for **printf**, the WPP supports predefined format specifications that can be used in trace messages. For example, %!NTSTATUS! represents a status value and displays the string associated with the status code. For a complete list of predefined format specifications, see the WDK.

Defining a Trace Message Function

The general format of a valid trace message function is:

FunctionName(*Conditions..., "Message", MessageVariables...*)

Conditions is a comma-separated list of values that appear before *Message*. (All parameters that appear before the message are interpreted as conditions.) The trace message is generated only if all conditions are true.

As for **DoTraceMessage**, *Message* specifies a string constant that defines the text of the trace message including format specifications, and *MessageVariables* specifies a comma-separated list of driver-defined variables whose formatted values are added to the message.

For example, in the following trace message function, the conditions are a trace level, a trace flag, and the subcomponent of the provider that is generating the trace message.

MyDoTrace(*Level*, *Flag*, *Subcomponent*, "*Message*", *MessageVariables*...)

In code, this function might look like the following:

```
MyDoTrace(TRACE_LEVEL_ERROR, VERBOSE, Network, "IOCTL = %d",
ControlCode);
```

where:

TRACE_LEVEL_ERROR is a standard trace level defined in *evntrace.h*.

VERBOSE is a trace flag defined by the provider.

Network is the subcomponent of the provider.

"IOCTL = %d" is the message string.

ControlCode is the variable to substitute for the **%d** format specifier in the message string.

To get the WPP to recognize your trace message function, you must:

- Write alternative versions of the macros that support the **DoTraceMessage** macro (**WPP_LEVEL_ENABLED** and **WPP_LEVEL_LOGGER**).
- Add a **-func** parameter that specifies your trace message function to the RUN_WPP statement that invokes the WPP.

For more information about these tasks, see the software tracing FAQ "Can I Customize **DoTraceMessage**?" in the WDK.

Converting Existing Debug Print Statements to Trace Messages

The RUN_WPP directive has numerous options for controlling the WPP. One commonly used option is **-func:***FunctionDescription*, which specifies an alternative to the **DoTraceMessage** function. The **-func** option can be used to "convert" existing debug print statements in a driver to trace messages. For example, suppose a driver already uses the following debug print command:

```
KdPrintEx((NULL, LEVEL, MSG, ...))
```

The following RUN_WPP directive would cause existing **KDPrintEx** calls to be treated as trace messages, in addition to any **DoTraceMessage** or other trace message function calls added to the driver source code:

```
RUN_WPP=$(SOURCES) -km -func:KdPrintEx((NULL, LEVEL, MSG,
...))
```

You can specify **-func** more than once in the same RUN_WPP command.

Running a Software Tracing Session

A software tracing session requires a trace provider and a trace controller, as well as a trace consumer to format and display the resulting trace messages.

This section describes how to run a software tracing session by using the *tracedrv* sample and the tracing tools Tracepdb, Tracelog, and Tracefmt to produce an ETL file that contains the messages generated by *tracedrv*. The examples use the *tracedrv* sample and the tracing tools provided in the Windows Server 2003 SP1 DDK.

Note: To use Tracepdb on versions of Windows earlier than Windows Longhorn, it is necessary to copy Dbghelp.dll file from the %WINDDK%\bin\Platform directory of the WDK into the directory in which Tracepdb.exe is located.

Preparing the Tracedrv Sample

Before starting a trace session with the *tracedrv* example, you must build both the sample driver and its test program, *tracectl*, then generate a TMF file and specify a path to the file so the Tracefmt tool can find and use this file to format the trace messages.

To prepare the *tracedrv* sample:

1. Use the WDK build utility to build *tracedrv* and *tracectl* from sample source code in %winddk%\src\general\tracedrv.
2. Use Tracepdb to generate a TMF file, using the following command.

   `tracepdb -f tracedrv.pdb -p` *path*

   The resulting TMF file is stored at *path*. Its name consists of a GUID with a .tmf filename extension.

3. Specify the path to the TMF file by setting the TRACE_FORMAT_SEARCH_PATH environment variable:

   ```
   SET TRACE_FORMAT_SEARCH_PATH=DriverPathName
   ```

The sample control GUID file *tracedrv.ctl* is included as part of the *tracedrv* sample files. If you write your own trace provider, you must create your own control GUID file. The file name is the name of the provider with a *.ctl* filename extension. The control GUID file is a text file that contains a single line consisting of the trace provider's GUID and its friendly name. For example, *tracedrv.ctl* contains the following line:

```
d58c126f-b309-11d1-969e-0000f875a5bc    CtlGuid
```

Running a Tracing Session with the Tracedrv Sample

Tracelog is a trace controller that runs at the command line. Tracelog can be used to enable, configure, start, update, and stop real-time and log sessions.

To run a tracing session by using Tracelog and the *tracedrv* sample:

1. Start the software tracing session by using the following Tracelog command:

   ```
   tracelog –start tracedrv –guid tracedrv.ctl
   –f tracedrv.etl –flag 0x1
   ```

   This command starts a trace session and associates the *tracedrv* sample with the session. The command also enables the *tracedrv* sample for tracing so it will generate trace messages.

   In this command:

   tracedrv is the name of the trace session.

   Tracedrv.ctl is the sample's control GUID file, which specifies its control GUID and a friendly name for the GUID.

   Tracedrv.etl is the name of the trace log file that receives trace messages from the session buffers.

   The **-flag** option is set to 0x1, which corresponds to the trace flag TRACELEVELONE specified in the *tracedrv* sample's WPP_CONTROL_GUIDS macro. Only trace messages with this flag are generated during this session. (Although *tracedrv* defines two trace flags, it implements messages only for TRACELEVELONE, so **-flag** must have a value of 1 for *tracedrv* to generate any trace messages.)

2. Run the sample test program *tracectl.exe*.

   Tracectl.exe loads *tracedrv* and, each time you type a letter other than q or Q, it sends an IOCTL to *tracedrv*. The IOCTL is the event that triggers trace messages. Type q or Q to quit *tracectl*.

3. Stop the tracing session by using the following Tracelog command with the name of the session provided in Step 1:

   ```
   tracelog –stop tracedrv
   ```

   This command flushes the session buffers to write any remaining trace messages to the trace log file, disables *tracedrv* so it no longer generates trace messages, and ends the trace session.

Viewing the Trace Log from the Tracedrv Session

Tracefmt is a command-line trace consumer that formats trace messages from real-time trace sessions or trace logs and writes them to files or displays them in a command window.

Tracefmt uses formatting instructions from the TMF file created earlier to convert the binary trace messages to readable form. The Tracefmt example shown here assumes that the TRACE_FORMAT_SEARCH_PATH environment variable is set to the path of the TMF file, as described earlier. If this variable is not set, specify a path to the file.

To view the trace log from the Tracedrv session:

1. Format the trace messages by using the following Tracefmt command:

   ```
   tracefmt –o tracedrv.txt –f tracedrv.etl
   ```

   In this command:

   Tracedrv.txt is the name of a text file to receive the formatted trace messages.

   Tracedrv.etl is the name of the log file containing the trace messages in binary format.

2. View the converted trace messages by using any text editor.

The following example shows the first few lines of a typical *tracedrv* sample log file, as formatted by Tracefmt:

```
EventTrace
[0]0D44.0CF8::03/02/2005-15:18:09.711 [tracedrv]IOCTL = 1
[0]0D44.0CF8::03/02/2005-15:18:09.711 [tracedrv]Hello, 1 Hi
[0]0D44.0CF8::03/02/2005-15:18:09.711 [tracedrv]Hello, 2 Hi
[0]0D44.0CF8::03/02/2005-15:18:09.711 [tracedrv]Hello, 3 Hi
[0]0D44.0CF8::03/02/2005-15:18:09.711 [tracedrv]Hello, 4 Hi
```

Tracefmt adds a default trace message prefix for the message logged by the corresponding **DoTraceMessage** call. The default trace message prefix specifies the CPU number, process ID, thread ID, timestamp in Greenwich mean time (GMT), and the control GUID friendly name. For information about formatting trace message prefixes, see the documentation for Tracefmt in the WDK.

Viewing Trace Messages in Real Time

Traceview is a trace controller and trace consumer with a graphical user interface. It is designed especially to display trace messages as they are generated. As a trace controller, Traceview can be used to enable, configure, start, update, and stop tracing sessions. As a trace consumer, Traceview can be used to format, filter, and display trace messages from real-time trace sessions and trace logs. TraceView combines and extends the features of Tracepdb, Tracelog, and Tracefmt. For information about using Traceview, see its online help.

To view trace messages in real time:

1. Start Traceview.
2. Create a new log session (on the **File** menu, click **Create New Log Session**).
3. Add *tracedrv* as a provider by selecting either its PDB file or its control GUID file (or paste its control GUID into the **Manually Entered Control GUID** box).
4. Specify the *tracedrv* TMF file, either by selecting the file or setting the TMF search path.

5. Select the **Real Time Display** check box and then click **Finish**.
6. Run the *tracectl* test program to load *tracedrv* and generate trace messages.

   The trace messages appear in the **TraceView** window, as shown in Figure 2.

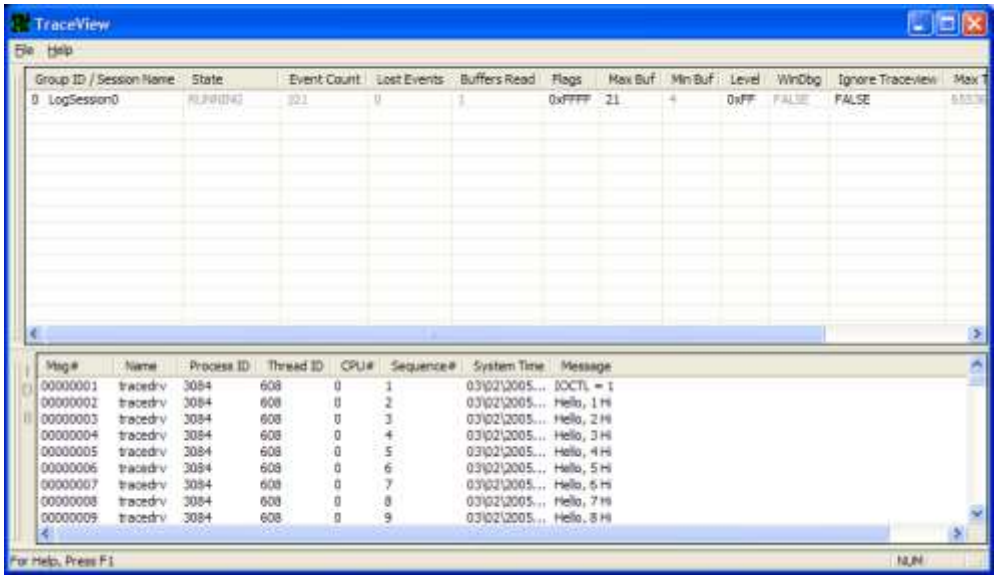

Figure 2. The TraceView window showing TraceDrv sample messages

Redirecting Trace Messages to a Kernel Debugger

Several methods can be used to redirect trace messages to a kernel-mode debugger. You can redirect trace messages to kernel debugger (KD) or WinDbg, whichever is attached. (You cannot redirect trace messages to other kernel debuggers, such as NTSD.)

To display trace messages in a debugger, *wmitrace.dll* and *traceprt.dll* must be in the debugger's search path on the host computer. Also, to enable the debugger to find the TMF files for the trace messages, you must either set the value of the %TRACE_FORMAT_SEARCH_PATH% environment variable or use the **!wmitrace.searchpath** debugger extension to indicate the location of the TMF files.

The **!wmitrace** debugger extension displays trace messages in the trace session buffers before they are written to log files or delivered for display. Debugging Tools for Windows contains **!wmitrace**.

Redirecting Trace Messages to KD

To redirect trace messages to the KD by using Tracelog:

1. Start KD.

   You must start the kernel debugger before submitting a Tracelog command with the **-kd** parameter. If KD is not running, Tracelog stops responding ("hangs").
2. In a command window, enter the following Tracelog command:

   ```
   tracelog -start MyTrace -guid MyProvider.ctl -rt -kd
   ```

   In this command:

   -start starts a trace session named MyTrace.

-guid specifies the trace provider's control GUID file. If this file doesn't exist, create it as described earlier in this paper.

-rt specifies that this is a real-time trace session.

-kd redirects the trace messages to the kernel debugger and sets the maximum buffer size to 3 KB, the maximum for the debugger.

3. Use the **!wmitrace** debugger extension to view the messages in the debugger.

For information about using **!wmitrace**, see the Debugging Tools for Windows documentation.

Redirecting Trace Messages to WinDbg

To redirect trace messages to WinDbg by using TraceView:

1. Start WinDbg, and then start TraceView.
2. Create a new trace log session, as described earlier in this paper.
3. After specifying the provider, the TMF file, and Real Time Display, click **Advanced Log Session Options**.
4. On the **Advanced Log Session Options** page, click the **Log Session Parameter Options** tab, and then change the value of the **WinDbg** option from **FALSE** to **TRUE** and click **OK**.
5. Click **Finish** to finish creating the trace log session.

For information about using WinDbg, see the Debugging Tools for Windows documentation.

Software Tracing Advances for Windows Vista

The Microsoft Windows Driver Kit (WDK) and Microsoft Windows Vista™ support several new features for the WPP software tracing.

- **Dynamic API selection.** Released versions of the WPP determine which logging functions to call at compile time, which requires compiling a separate binary for Windows 2000. For Windows Vista, the WPP determines which logging functions to call at run time, so a single binary can run on Windows 2000 and still take advantage of new and improved tracing functions in later versions of Windows.

  Usage for the WPP macros does not change. However, when **DoTraceMessage** and similar macros generate trace messages, they call an appropriate function for the operating system on which the provider is running. Specifically, when a WPP-enabled component runs on Windows 2000, **DoTraceMessage** calls **TraceEvent** in user mode and **IoWMIWriteEvent** in kernel mode. When the same WPP-enabled component runs on Windows XP or Windows Server 2003, **DoTraceMessage** calls **TraceMessage** in user mode and **IoWmiTraceMessage** in kernel mode. This makes it possible to write a single provider that runs on Windows 2000 and later without requiring the use of special template files.
- **New pre- and post-logging macros.** Pre- and post-logging macros define the WPP_LEVEL_PRE(level) and WPP_LEVEL_POST(level) macros, which enable user code to become part of the tracing function’s macro expansion. Customers can use this for dynamic setup or cleanup around trace points in the field.
- **Support for enumerated types.** Enumerations can be used to display meaningful names in trace messages, instead of displaying integer values that

users must decode. To use an enumeration in trace messages, add configuration data such as the following example to your source code file.

```
// begin_wpp config
// CUSTOM_TYPE(FormatName ItemEnum(EnumName) );
// end_wpp
```

In this example, *EnumName* is the enumeration and *FormatName* is the custom type that can be used in the format string of a trace message. To notify the WPP to look for this configuration data in the source code file, add a **-scan:***sourcecodefile* parameter to the RUN_WPP macro that invokes the WPP.

- **Tracewpp module name parameter.** The new Tracewpp command-line argument **p:***modulename* replaces the default <*directory*> string with a more convenient user-defined message. This helps connect files that are part of one logical module but in separate directories and helps distinguish among files that are in separate modules but have visually similar paths.
- **New TMF locator feature in Tracefmt.** The TMF locator feature enables the Tracefmt tool to locate a provider's PDB file by using the image file that created the trace and the symbol server or symbol path.
- **New !wmitrace debugger extension settings.** The new TRACE_FORMAT_PREFIX command can be used to specify the tracing prefix during a debugging session.

  The TRACE_FORMAT_SEARCH_PATH command has been enhanced to allow adding directories to the existing search path during a debugging session. Previously, TRACE_FORMAT_SEARCH_PATH permitted setting the entire path but not retrieving the path and adding to it.

Summary of Tools for Software Tracing

The WDK includes a set of GUI-based and command-line tools for software tracing. These tools are designed to support ETW and to supplement the tracing tools that are included in Windows.

Tracing Tools in the WDK

The WDK tracing tools are located at %winddk%\tools\tracing\. They include:

- **Tracelog**, a command-line trace controller that enables, configures, starts, updates, and stops real-time and log sessions. Tracelog supports user-mode and kernel-mode trace sessions, NT kernel logger trace sessions, the global logger (boot) trace session, and tracing to measure time spent in deferred procedure calls (DPCs) and interrupt service routines (ISRs).
- **Tracepdb**, a command-line support tool that creates TMF files from PDB symbol files. TMF files are required input to Tracefmt and can be used in TraceView.
- **Tracefmt**, a command-line trace consumer that formats trace messages from real-time trace sessions or trace logs and writes them to files or displays them in the **Command Prompt** window.
- **TraceWPP**, a command-line tool that runs the WPP on the source files of trace providers. An alternative to running the WPP in the WDK Build utility, TraceWPP processes trace macros in a source file and creates a header file to enable the WPP tracing. TraceWPP (tracewpp.exe) is included in the WDK in

the \bin\x86 directory. The command-line options for TraceWPP are the same as those for RUN_WPP.

- **TraceView**, a GUI-based trace controller and trace consumer designed especially for real-time displays of trace messages. It enables, configures, starts, updates, and stops tracing sessions, and formats, filters, and displays trace messages from real-time trace sessions and trace logs. TraceView combines and extends the features of Tracepdb, Tracelog, and Tracefmt.

Tracing Tools Included with Windows

Tracing tools included with Windows XP and later versions of Windows include:

- **Logman**, a fully functional, GUI-based trace controller designed especially 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 CSV or XML files. Tracerpt also analyzes the events and generates summary reports.

Note: The Event Viewer is a Control Panel application that displays events recorded in various application, system, and security logs. It is intended primarily for end users to monitor basic system events such as errors loading system services. The Event Viewer cannot be used to display trace logs generated by drivers.

Summary of Tracing Functions and Macros

Tracing Functions and Macros for Kernel-Mode Drivers

For information about these tracing functions and macros, see the WDK.

WPP Trace Provider Macros

WPP_CONTROL_GUIDS
WPP_INIT_TRACING
WPP_CLEANUP
DoTraceMessage

WPP Preprocessor Directive

RUN_WPP

WMI Trace Provider Functions

IoWmiRegistrationControl
IoWMIWriteEvent with WNODE_FLAG_TRACED_GUID
WMiTraceMessage (Windows XP and later)
WmiTraceMessageVa (Windows XP and later)

WMI Trace Collection Control Function

WmiQueryTraceInformation

Tracing Functions for User-Mode Applications

For information about these functions, see the Platform SDK documentation.

Trace Provider Functions

- **TraceEvent**
- **TraceEventInstance**
- **RegisterTraceGuids**
- **UnregisterTraceGuids**
- **TraceMessage**
- **TraceMessageVa**

Trace Consumer Functions

- **OpenTrace**
- **CloseTrace**
- **ProcessTrace**
- **SetTraceCallback**
- **RemoveTraceCallBack**

Trace Collection Control Functions

- **StartTrace**
- **StopTrace**
- **QueryTrace**
- **EnableTrace**
- **UpdateTrace**
- **QueryAllTraces**

Resources

References

Platform SDK:

Event Tracing
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/perfmon/base/event_tracing.asp

WHDC:

Event Tracing
http://www.microsoft.com/whdc/devtools/tools/EventTracing.mspx

OSR Online:

WPP Tracing Part 1 – Supporting Windows 2000 and Beyond
http://www.osronline.com/article.cfm?id=376

WPP Tracing Part II – Coexisting Peacefully with WMILIB
http://www.osronline.com/article.cfm?article=377

Kits and Tools

Windows DDK:

http://www.microsoft.com/whdc/DevTools/ddk/default.mspx

Debugging Tools for Windows:

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

Windows Driver Kit (WDK):

Preview versions of the WDK are available through the Vista Beta program.
http://www.microsoft.com/whdc/driver/ldk/default.mspx