Interactive Session: Software Tracing

Ian Service
Software Design Engineer
Reliability
lanserv@microsoft.com

Contents

- Overview
- Recap on what Software Tracing is
- How Software Tracing works
- Build Environment
- Using your driver with Tracing
- Customizing Tracing
  - Formatting
  - Trace Macros
- Debugging Tracing
- Tracing and the Kernel Debugger
- Questions and hands on exercises with the demo driver or own code if preferred

Overview

- This session/lab will provide an overview of software tracing, how it works, and an introduction to some of its advanced features; The session will also cover how you can use tracing as a problem-solving tool on production systems, the advantages of common levels and standards for tracing, and guidelines on writing effective trace messages
  - Using the standard DDK examples to show the use of the different tools
  - Demonstrating the differences in trace capability on different versions of Windows
  - Converting an existing driver with debug calls to tracing when on the free build
  - Mixing different trace sources, multiple drivers, and mixed user mode and driver traces
  - Making use of different trace log options
  - Debugging tracing problems
  - Using the kernel debugger tracing extension for live and crash dump access to traces
  - Controlling tracing by using different parameters, for example, levels instead of bit flags
  - Adding new types to tracing
  - Making use of hex dumps and other special formats in tracing
  - Mixing software tracing with other tracing

Recap On “What Is Software Tracing?”

- Assumption is everyone knows about “Software Tracing”
- Software Tracing is
  - An alternative to using a “checked build” by including “debug prints” I in the released version of the code
  - Efficient when generating traces, and minimal overhead when not enabled
    - Minimal HeisenBug effect
  - Selective, in that it may be enabled on a component basis and may be enabled at different levels
  - Dynamic in that it can be enabled or disabled without stopping/restarting a component, and especially without rebooting the OS
  - Flexible, as it can adapt and take advantage of existing instrumentation
  - Easy to implement, as automated as possible on the ground that “it has to be as simple as printf, or developers won’t use it”

Recap On “What Is Software Tracing”

- Software Tracing Comprises
  - A preprocessor which extracts trace entries from source files and creates macro’s to perform tracing
    - These Macros generate code that
      - Store trace decoding information in the PDB
      - controls tracing with your driver
      - Marshal’s your trace arguments and sends them to a logger
  - A system concept known as a logger which accepts your trace information and “logs it” appropriately
    - Key point is that the “where” is not in the callers code. Trace output can be redirected to different destinations without a code change
  - Viewer programs which accept the logged output and combine it with the decode information, and display it
    - Key point is that all the formatting is done after the fact, not in the execution path

Demonstration

- TraceDrv is a simple example that's inside the DDK
- We will use this as an example at several points
  - To save demonstration time we will also use the simple application "DocExample"
    - It is user mode but the features of user mode tracing are identical
- TraceDrv can be built to be Windows XP and later or Windows 2000 and later compatible
  - Sources file is a good example of this
  - However for best Windows 2000 operation I strongly recommend the use of WDF
- We will build TraceDrv and execute it
  - Let's go through the whole process with TraceDrv
    - Build
    - Execute
    - Trace

Build Environment

- Next few slides will give an explanation of what is going on in the build environment as regards tracing
- A view of the individual files online with examples to show the significant parts
  - Sources file
    - This is how we make sure that tracing is invoked for the project
    - Makefile rules for tracing
  - TraceWPP – the Trace PreProcessor
    - What it does and how it can be controlled
  - Build tool changes
    - Binplace and how placefil.txt is used

Tracing And The Build Environment – Sources File

- The Sources file is the key file for defining how to build your project
- RUN_WPP is the function used in generating software tracing
  - Simplest form is
    - RUN_WPP=$(SOURCES)
  - Usually that's all that is needed, but there many options
    - See the TRACEWPP slide for detailed options
- Under the covers
  - Makefile.def has some rules that cause TraceWPP to be executed before anything else
    - Sets up to use the default templates (TPL files) as input
    - Defines the compile variable RUN_WPP
    - Generates TMH files

Tracing And The Build Environment – TraceWPP Options

- Tracewpp is the simple preprocessor that runs as part of the build environment
  - Input: Source file(s) plus template files
  - Output: Trace Message File(s)
- Table shows all of the significant options
  - Mostly the defaults are fine, for driver writers “–km” is always required

Option	Description	Build Default	Default
-odir:path	Path for TMH files	$(O)	Current dir
-km	Tracing for a kernel component	User Mode	User Mode
-dll	Tracing for a DLL	Not a DLL	Not a DLL
-cfgdir:path	Path for the config and template files	DDK template path	Current dir
-ini:path	Specify a extra config file	None	None
-ext:.ext1.ext2	Types of files to scan	.c.c++.cpp .cxx	.c.c++.cpp .cxx
-scan:file	Scans the file for config. info	None	None
-func:desc	Specifies the trace function	See localwpp	See Localwpp
-defwpp:path	Override for localwpp.ini	Config path	Config Path

Tracing And The Build Environment – More Tracewpp Options

- Some extra options
  - Just for completeness

Option	Description	Build Default	Default
-v4	Outputs trace operation	None	none

Tracing And The Build – Templates (TPL files)

- Template files are the key to generating the Trace Message Headers (TMH) files
  - TPL file + your trace statements → TraceWPP → TMH file
  - TPL files simply define the c/c++ macros to be created
- Defaultwpp.ini – this defines all the basic types that tracing supports
  - We will come back to this later to show how to add types
- Km-default.tpl
  - Is the first template that is scanned when you build a driver
    - Invoked by the –km switch
    - If u use Windows 2000 mode its actually km-w2k.tpl that is called first
  - These “includes” all the other templates

Tracing And The Build – Trace Header Files (.TMH Files)

- Trace Message Header (.TMH) files
  - Each source which includes tracing has a #include “file.tmh”
    - Where file is the same file as the source, without the extension replaced
  - This defines all the macros for this particular source module
    - For example every trace statement in the source file will have a line in the trace macro defined within “file.tmh”
      - Each identified by line number inside a large macro
- Let's look a simple Trace statement
  - DoTraceMessage(Unusual, "Hello, %s %d", "World", i);
- What does it expand into logically
  - If (“Unusual” is enabled) {

    Add format information to the PDB

    Call WmiTraceMessage(and pass in the arguments)

    }
- And let’s look at the same statement in the TMH file

Tracing And The Build – Trace Message Files (.TMF Files)

- Trace Message Files (.TMF files)
  - These are the the decoder files for trace logs
    - This is actually the information that is contained in the PDB, but extracted so it can be used directly
      - There is one (or more) TMF file per source
      - There is also a TMC file which reflects the control Guid information
  - At the end of a Build command the tool binplace runs
    - Looks for your driver in placefil.txt
    - If it finds it, it
      - Places your driver executable in the designated directory
      - Places your full symbols in Symbols.Pri
      - Places your stripped symbols in Symbols
      - Places your TMF files in Symbols.Pri\TraceFormat
        - This is ONLY if its in placefil.txt
        - As an alternative tracepdb –f driver.pdb will create the TMF files
  - Let's take a look at the TraceDrv TMF files

How Tracing Works – Internals

- Explain Event Tracing for windows (ETW)
  - Introduced in Windows 2000
    - Win32 APIs and Driver APIs
    - Uses
  - ETW core concepts
    - Loggers
      - Explain how logger abstraction works
      - Limits, and types of loggers
  - Explain relationship of tracing to loggers
  - Loggers for drivers
  - Loggers for drivers and applications
- Online use the simple commands to see the system operation

How Tracing Works – Win32 APIs

- Just for completeness the relevant user mode APIs are
- Collection Control APIs
  - StartTrace(), StopTrace(), QueryTrace(), EnableTrace(), UpdateTrace(), QueryAllTraces()
- Trace Consumer API's
  - OpenTrace(), CloseTrace(), ProcessTrace(), SetTRaceCallback(), RemoveTraceCallBack()
- TraceProviderApi's
  - TraceEvent(), TraceEventInstance(), RegisterTraceGuids(), UnRegisterTraceGuids(), TraceMessage(), TraceMessageVa()

How Tracing Works – Driver Functions

- Functions which are relevant to tracing, that driver may use
- Collection Control APIs
  - WmiQueryTraceInformation()
- Trace Provider APIs
  - IoWmiRegistrationControl(), IoWMIWriteEvent*(), WMiTraceMessage**(), WmiTraceMessageVa** ()

* Used with the WNODE_FLAG_TRACED_GUID to distinguish it from other WMI usage

** Windows XP and later only

How Tracing Works – Standard Kernel Logger

- The operating system has some built in logging
  - Started in Windows 2000 has been extended in every release.
  - Process Start/Stop, Thread Start/Stop, Registry, File operations, disk operations, network operations, etc.
- This by itself can sometimes be useful to you as a developer
  - Combined with your tracing it can be even more useful
- The decoder file (system.tmf) is in the tools\tracing directory in the DDK
  - Take a look inside, it's a special case TMF files but the format is the same

Demo/Hands on trial

Run Traceview -> Add Provider –> Kernel Logger

How Tracing Works – Tools

- Logman
  - Standard control tool, starts stops traces, etc.
  - Runs on Windows XP and later, ships “in the box” in Windows XP Pro and onward
    - In SP2 will be added to Home edition
- Tracelog
  - Original prototype control tool
  - Runs on Windows 2000 and later (But recommend logman for non-Windows 2000 systems)
  - Available in the support CD, DDK and SDK
    - Source of the original version In the SDK
      - Good information on how to control traces
- TraceFmt
  - Command line trace formatting tool
  - Relies on traceprt.dll
  - Available in the support CD, DDK and SDK
- Tracepdb
  - Extracts Trace Message Information from PDBs, creates TMF files
  - Available in the support CD, DDK, and SDK

How Tracing Works – Tools Continued

- Traceview
  - GUI based trace viewer
    - Includes functionality for control and for extracting TMF information
      - “One stop shopping” for developers
    - Uses Traceprt.DLL
    - Ships in the DDK
- WmiTrace KD extension
  - Works with the debugger to display traces from the system memory
    - Live, or from crash dumps.
  - Uses traceprt.dll
  - Ships with the debugger

Using Your Driver With Tracing

- Thinking about when/where to add trace points
- When to release your trace decode files
- Levels of tracing
  - Tracing adapts to different styles
  - But a common model can be helpful
- Combining tracing from different components
  - Drivers and user components for example
- Online Exercise
  - Run Tracedrv and the Trace Control Program to the same log file

Using Tracing – Standard levels

The definitions from SDK\inc\evntrace.h

```
//
// Predefined Event Tracing Levels for Software/Debug Tracing
//
//
// Trace Level is UCHAR and passed in through the EnableLevel parameter
// in EnableTrace API. It is retrieved by the provider using the
// GetTraceEnableLevel macro. It should be interpreted as an integer value
// to mean everything at or below that level will be traced.
//
// Here are the possible Levels.
//

#define TRACE_LEVEL_NONE        0   // Tracing is not on
#define TRACE_LEVEL_CRITICAL    1   // Abnormal exit or termination
#define TRACE_LEVEL_FATAL       1   // Deprecated name for Abnormal exit
#define TRACE_LEVEL_ERROR       2   // Severe errors that need logging
#define TRACE_LEVEL_WARNING     3   // Warnings such as allocation failure
#define TRACE_LEVEL_INFORMATION 4   // Includes non-error cases(e.g.Entry-Exit)
#define TRACE_LEVEL_VERBOSE     5   // Detailed traces from intermediate steps
#define TRACE_LEVEL_RESERVED6   6
#define TRACE_LEVEL_RESERVED7   7
#define TRACE_LEVEL_RESERVED8   8
#define TRACE_LEVEL_RESERVED9   9
```

Debugging Tracing

- What to do when your tracing doesn't do what you expect
  - Is it your code
    - Well that would be the last thing to suspect
  - Or is it Tracing
    - None of us fully trust code that is generated by anyone else do we?
- We often get asked the same thing and so we have a few techniques that help us to resolve issues

Debugging Tracing – Using Internal Debugging Features

- Use of internal Tracing debug features
  - Trace state changes
    - This is the most convenient to figure out if your tracing is being enabled or the flags you expect are being set
      - Incorrect GUID's or wrong levels/flags are very common
    - #define WppDebug(a,b) DbgPrint b
      - Or any other flavor of print you prefer
      - This causes the tracing subsystem to output some status messages on control transitions
  - Debug Prints
    - This is a convenient way to tell if your trace statements are malformed
    - #define WPP_DEBUG(a,b) DbgPrint b
      - Or again any flavor of print you prefer
      - This causes all your trace statements to be output using the debug prints selected
        - NB if you use custom trace formats this does not work well

Debugging Tracing – Other Techniques

- Well was the code that Tracing generated correct?
  - Two good ways to check
    - Viewing the actual code in a pre-processor file file
      - Use nmake file.pp to generate it
  - Viewing TMH files
    - We looked at these earlier but if you are really puzzled well, all the information is here
- Let’s take a quick look at these files online

Customizing Tracing – Formatting The Trace Output

- Tracing has a standard format, and usually we recommend that it be used
  - Allows different traces to be viewed together
- Sometimes it is convenient to modify it, and there are several mechanisms
- Build time
  - Redefine the prefix, or add a suffix
- Format time
  - Override the prefix, or suppress it entirely

Customizing The trace Output – Using A Prefix/Suffix

- By default all trace statement have a
  - Standard Prefix, known as “%0” which stands for
    [CPU#]ProcessID.ThreadID::Timestamp [FileName_line]
  - A null Suffix
- A good example of all of these is the Function Entry/Exit tracing
- Here is an example of the code you might write

```
#include "mytrace.h"
#include "example2.tmh"
examplesub(int x)
{
    FuncEntry();
    // do some real work
    FuncExit();
}
```

Customizing The Trace Output – Using A Prefix/Suffix

- Mytrace.h is the example file

```
#define WPP_CONTROL_GUIDS \
    WPP_DEFINE_CONTROL_GUID(CtlGuid,(a044090f,3d9d,48cf,b7ee,9fb114702dc1),  \
      WPP_DEFINE_BIT(FuncTrace) )

// begin_wpp config
// FuncEntry();
// FuncExit();
// USESUFFIX(FuncEntry, " Entry to %!FUNC!");
// USESUFFIX(FuncExit, " Exit from %!FUNC!");
// end_wpp

// Map the null level used by Entry/Exit to TRACE_LEVEL_VERBOSE
#define WPP__ENABLED() WPP_LEVEL_ENABLED(FuncTrace) WPP_CONTROL(WPP_BIT_ ## FuncTrace).Level >=
        TRACE_LEVEL_VERBOSE
#define WPP__LOGGER() WPP_LEVEL_LOGGER(FuncTrace)
```

- **Then you make your run_wpp line in sources look like following**

```
RUN_WPP=$(SOURCES) -scan:mytrace.h
```

Customizing Tracing – Converting Your Debug Prints

- Probably everyone already has some sort of debug print code, likely organized in a way that they regard as perfect
  - Can we keep that code and use it as debug prints in the checked build – Yes
  - Can we convert it to tracing in the free build – Yes
- Let's assume you use KdPrintEx

```
ULONG KdPrintEx ( (
 IN ULONG ComponentId,
 IN ULONG Level,
 IN PCHAR Format,
  . . . [arguments]
) ) ;
```

- Which is fairly similar to the normal trace macro, with the addition of that ComponentID
  - Let's treat the component ID as NULL

Customizing Tracing – Example Converting KdPrintEx

1. Get a GUID for the component using guidgen or uuidgen (see note b)
2. Define your debug flags and GUID in a convenient header file
3. For each file with a debug print include that header file and a file called filename.tmh

   ```
   #ifn DBG
   #include “MyTracefile.h”
   #include “filename.tmh”
   #endif
   ```

4. Add a WPP_INIT_TRACING call in your DriverEntry routine

   Wrap in a #ifndef DBG conditional so it is not used by checked build
5. Add a WPP_CLEANUP call to your DriverUnload routine

   Wrap in a #ifndef DBG conditional so it is not used by checked build
6. In the SOURCES file add

   RUN_WPP=$(SOURCES) –km –func:KdPrintEx((NULL,LEVEL,MSG,…))

   Wrap in an ! If !(FREEBUILD) so it is not called by the checked build

Note:

a. Component ID is NULL’ed out in the “func” prototype

b. Really we don’t need the Component ID anyway as the Component GUID replaces it

Customizing Tracing – Special Trace Formats

- Software Tracing has a number of special formats built in
  - Downside to these is they DO NOT work if you rebuilt your trace statements as debug prints
    - So use carefully
  - In general they have the form %!name!
    - STATUS - prints out a value as the NTSTATUS name
      DoTraceMessage("NTSTATUS is %!STATUS!\n",ntStatus);
    - WINERROR - prints out a value as the WinError name
      DoTraceMessage("WINERROR is %!WINERROR!\n",myerror);
    - HRESULT - prints out a value as the HRESULT name
    - IPADDR - prints out a value as the IP (v4) style Address
    - GUID - prints out a value formatted as a GUID format

Customizing Tracing – Macros

- Trace macros cannot use the C pre-processor – how to customize
  - Show how to define prefix/suffix
  - Show how to define PRE and POST macros
- To demonstrate: Modify example to add an ASSERT style trace macro

Tracing And The Kernel Debugger

- There are two ways to use the kernel debugger with tracing
  - Dynamic tracing, similar to DbgPrint
    - But not the same!
  - Post Mortem Tracing
- Settings
  - For all of this the kernel debugger used the KD extension wmitrace.dll
    - Sort of acts like tracefmt/traceview and uses traceprt.dll
  - Before starting the debugger environment variables should be set
    - Set TRACE_FORMAT_SEARCH_PATH=path
      - This can be done inside the extension
  - Tracelog/Logman settings
    - “-kd” for direct delivery of trace buffers to the debugger
    - “-rt B –age -1” for efficient in memory buffering for post mortem use

Tracing And The Kernel Debugger

- Limitations
  - Trace buffers between the target and the debugger host are limited to 3K in dynamic tracing
  - Dynamic Tracing is buffered so it does not synchronize with real Debug prints
- When to use
  - When you need to debug without having a debug version
  - Bugs go away when you use debug prints
    - Post mortem taces can help
  - But this is NOT a replacement for dbgprint!
    - It is just another tool for you