Using the ETW tracing preprocessor in your Application

Table of Contents

What is this ETW thing?

Event Tracing for Windows is a kernel-level tracing service that has been around since Windows 2000. Since it's baked right into the kernel, it is extremely fast. Most of the developers that use ETW are writing drivers, but why should they have all the fun?

Why should I use ETW?

ETW Tracing has several benefits over the tracing classes provided with the .NET Framework. Most importantly, ETW tracing can be turned on and off without having to restart the application, but it also has features like built-in high performance circular logging (a circular log is one that never grows above a specified size by flushing out older trace messages), and the ability for you to capture the logs from multiple sources into a single trace session.

What is this preprocessor and why do we need it?

Put simply, to maximize application performance when tracing is not enabled. In a perfect world, an application's performance when tracing is disabled would be identical to one where tracing wasn't included at all. The problem is that your code is only compiled once; if those trace calls are in there, they're GOING to get called, and while the ETW functions return quickly when tracing is disabled, the runtime still has to evaluate trace arguments, allocate memory, construct method call stacks, and so on. The application performance would be even faster if the functions were never called in the first place.

How much faster is it? Here's an example: Let's write a simple application that has a function named DoSomething.

```
static int DoSomething(String arg0, int arg1, long arg2, DateTime arg3)
```

As you can see, DoSomething in this case simply returns a value and does no other calculations. It should be blazingly fast, right? Well, it is, but there's still the overhead of the method call. To demonstrate this, let's run two loops: one that ends up in a call to DoSomething one million times, and another that will shortcut the call. To do this, we'll create a method named DoRun that will call DoSomething unless the caller has specified that it should bypass the call entirely. If the value passed to shortcut is true, we'll skip the DoSomething call altogether.

```
static void DoRun(bool shortcut)
{
  DateTime start, stop;

  start = DateTime.Now;
  for (int index = 0; index < 1000000; index++)
  {
    if (!shortcut)
    {
      Program.DoSomething("Hi there!", 42, 42L, DateTime.Now);
    }
  }
  stop = DateTime.Now;

  Console.WriteLine(
    "Shortcut {0}: {1} milliseconds",
    shortcut,
    (stop - start).TotalMilliseconds);
}
```

To get our results, we will now call DoRun twice; once with shortcut set to False, and again with shortcut set to True. On my machine (a 2GHz Core2Duo running Windows Vista x86), the results are:

```
Shortcut False: 546 milliseconds
Shortcut True: 15.6 milliseconds
```

In other words, it was 35 times faster to completely bypass the function call. That's nearly two orders of magnitude! Now, imagine if you were doing something even more complicated there such as calling ToString() on an exception or dumping a string containing the values of all of the properties of the object you are working on and you should be able to see that you get a rather large performance boost by skipping those calls completely.

The NTrace API

The NTrace API is actually quite simple: there is a single class named EtwTrace with a method named Trace. This method can take an optional trace level to indicate the "severity" of a message (e.g.: Verbose, Information, Warning, etc.), an optional bit-flag that can be used to indicate a functional "area" of tracing (e.g.: Disk I/O, Workflow activities, network communications, etc.), and a format string + arguments similar to that of String.Format. In fact, the format of the format string used for NTrace was deliberately lifted straight from String.Format so that prospective developers wouldn't have to learn yet another new (well, technically, old, but that's another story) set of formatting rules.

Using NTrace in New Projects

If you're one of the lucky few who never need to maintain existing applications, then you can leverage the project templates that ship out of the box with NTrace. As of the time this document was written, there are currently templates for C# Console applications, class libraries, and WinForms applications with plans for project templates for WCF, ASP.NET, and WPF applications soon. Starting a project using the project templates sets everything up for you: all you need to do is write your code and add tracing calls.

Adding NTrace Support to Existing Projects

Declare an ETW Provider

In order to have your application publish events to ETW, you must add a reference to the NTrace library to your project (in the %ProgramFiles%\NTrace folder or in your machine's GAC) and include a static field that is of type NTrace.ClassicProvider. The name you use for the variable isn't really important, but the fact that it can be accessed by all of the classes in your assembly is. In practice, it is advised to use internal so that it is accessible from all of the classes in your assembly.

Tweak An Existing Project

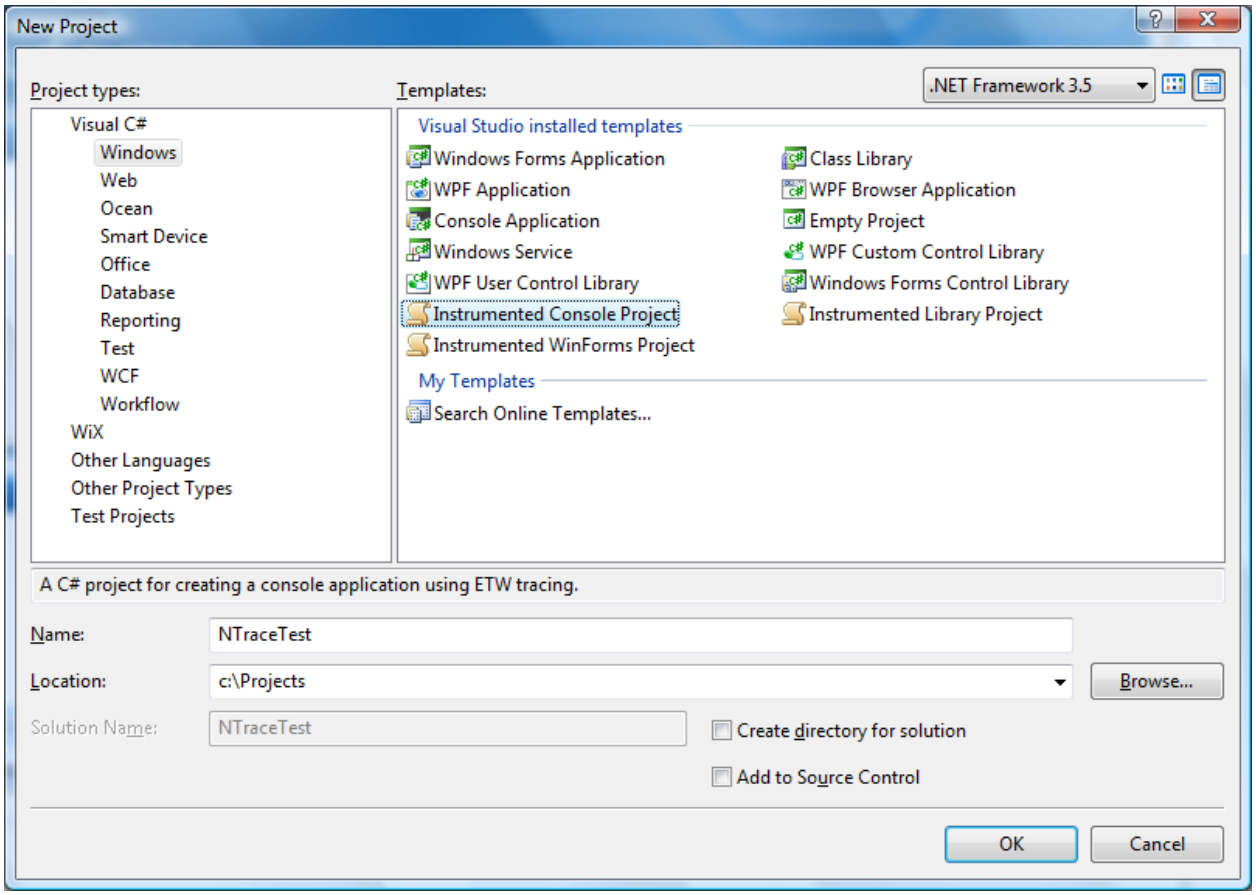
In many cases, you'll want to add the preprocessor to your existing code. In order to have the preprocessor kick in and do its magic, you'll have to import the build instructions into your C# project. To do this, simply add properties registering your ETW provider and replace your existing reference to Microsoft.CSharp.targets with one to NTrace.CSharp.Targets from the $(MSBuildExtensionsPath)\NTrace\<<version>> folder like so:

```
  <!–
  These values "register" your provider and tells the preprocessor how
  To hook things up when it's time to build.
  -->
  <PropertyGroup>
    <EtwProviderVariable>
      <<fully-qualified provider variable name>>
    </EtwProviderVariable>
    <EtwProviderId>96B47F82-971F-4644-821F-B55FB2439DAD</EtwProviderId>
  </PropertyGroup>

  …

  <!–There will already be an Import at the bottom of your project, pointing
to Microsoft.CSharp.targets. Change it to the following -->
  <Import
Project="$(MSBuildExtensionsPath)\NTrace\v1.0\NTrace.CSharp.targets" />
```

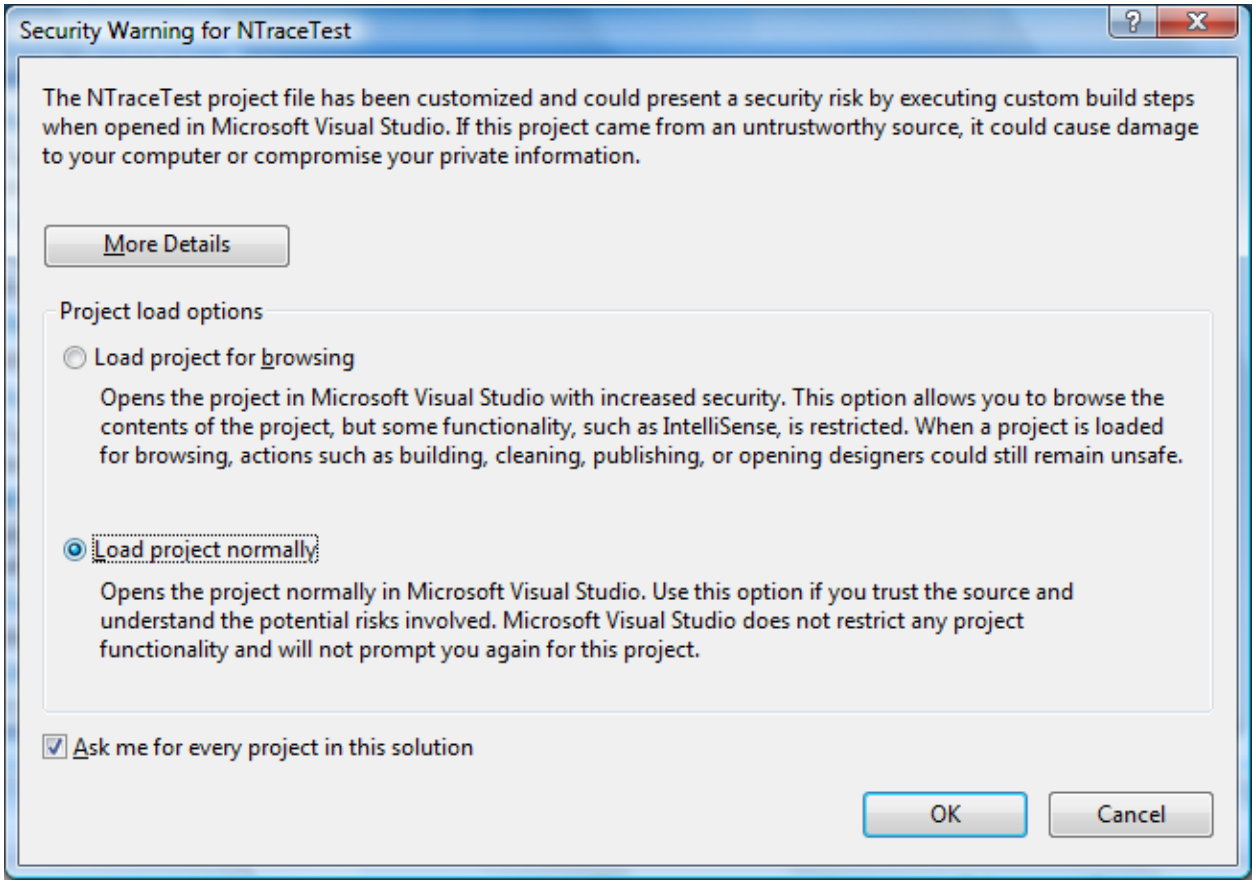
NTrace in Action

Creating a New Project

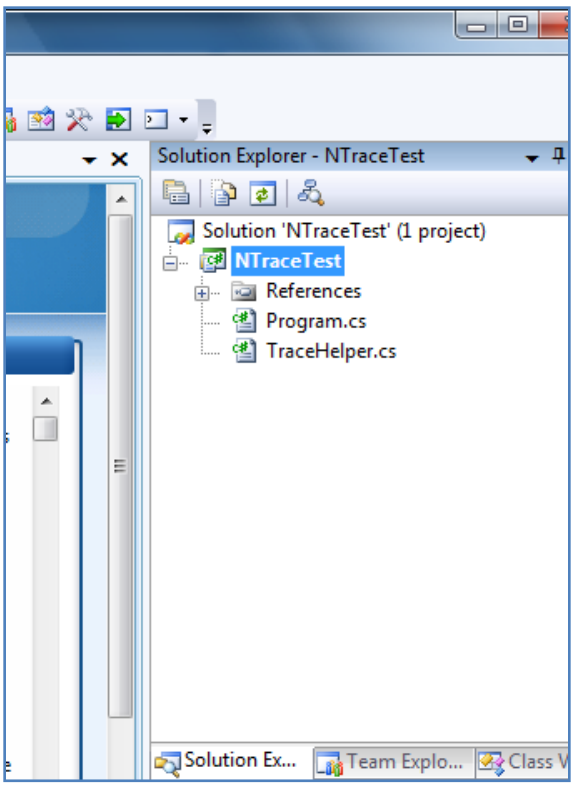
Let's walk through a demonstration. First, install NTrace. This will install the tracing library, the preprocessor, and Visual Studio 2008 project templates. Once everything's installed, open Visual Studio and create a new Instrumented Console Application project:



This will create the project and add the necessary assembly/MSBuild references. Due to an unfortunate bug in the current bits, you will be presented with a dialog informing you that the project has been customized. Until we can fix this issue, the first time you load an NTrace-enabled project, you'll see this dialog:



When you are presented with this dialog, simply choose “Load project normally” and click OK. Visual Studio will then complete the project creation process. At this point, you will have a new project (in this case, named ‘NTraceTest’):



At this point, we can edit Program.cs and add some tracing! As we mentioned before, an NTrace call looks like EtwTrace.Trace(…), so let's add a trace message that logs the classic "Hello, World!" message:

```
using System;
using System.Collections.Generic;
using NTrace;

namespace NTraceTest
{
  class Program
  {
    static void Main(string[] args)
    {
      EtwTrace.Trace("Hello, world!");
    }
  }
}
```

At this point, we're ready to compile and run. Assuming everything is working properly, you've got an application that is now using NTrace to log messages. Or so you hope, at least – you certainly aren't seeing anything to indicate that anything is really happening yet. So, let's prove to ourselves that we've actually succeeded.

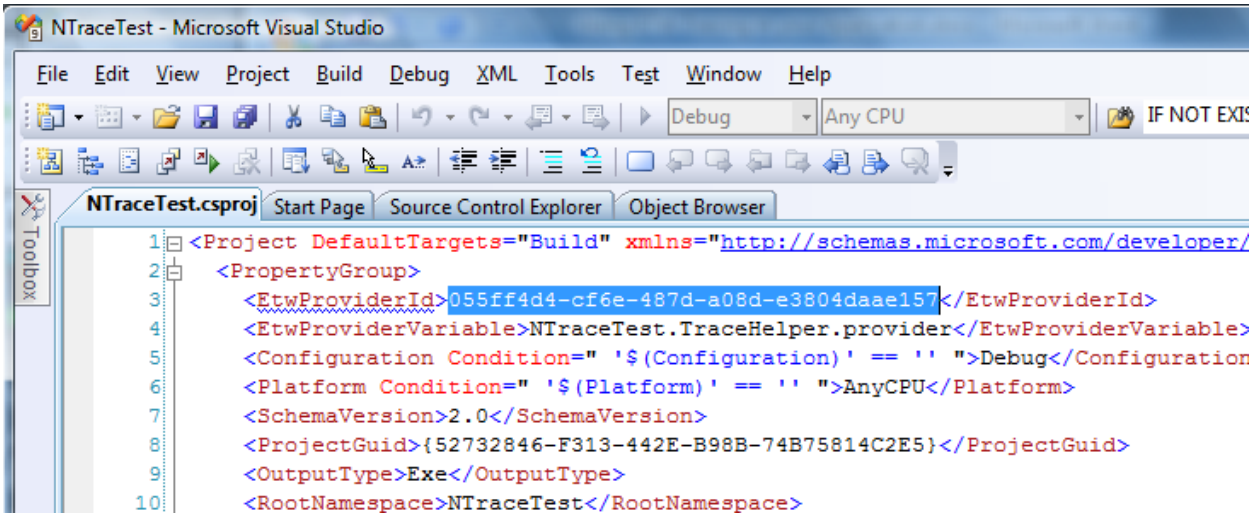
Using TraceView to Start a Tracing Session

NTrace uses the same tools that WPP tracing uses to capture trace logs: traceog.exe starts and stops tracing sessions, tracefmt.exe transforms the binary logs that ETW generates into human-readable text,

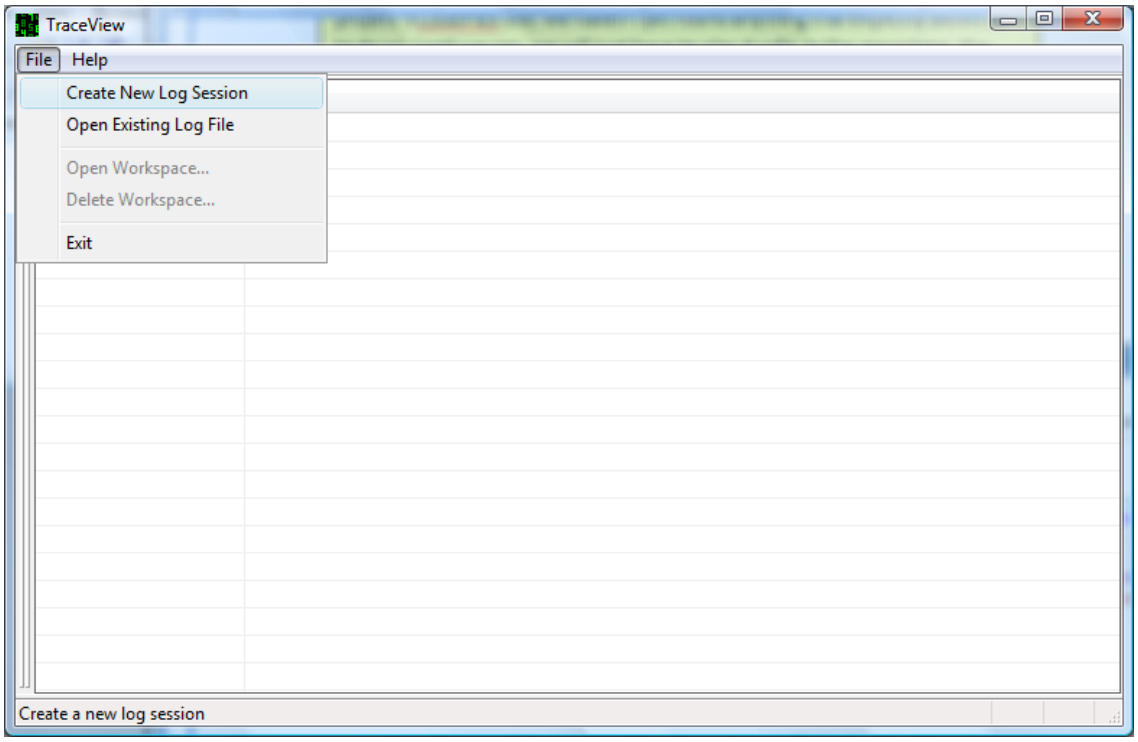
and TraceView.exe is a GUI that performs the same functions as the previous two command-line apps. In this case, let's use TraceView to start a new real-time logging session.

> **Sidebar: Installing the Tracing Tools**
>
> While we'd love to serve up the binaries for tracelog.exe, et.Al. from the project's CodePlex site, we haven't yet found anything that explicitly allows us to do so. Until we can, we will just have to play it safe. In the meantime, the easiest way to get these tools is to download the Windows 2003 DDK, and just install the tracing tools:
> http://www.microsoft.com/whdc/DevTools/ddk/default.mspx

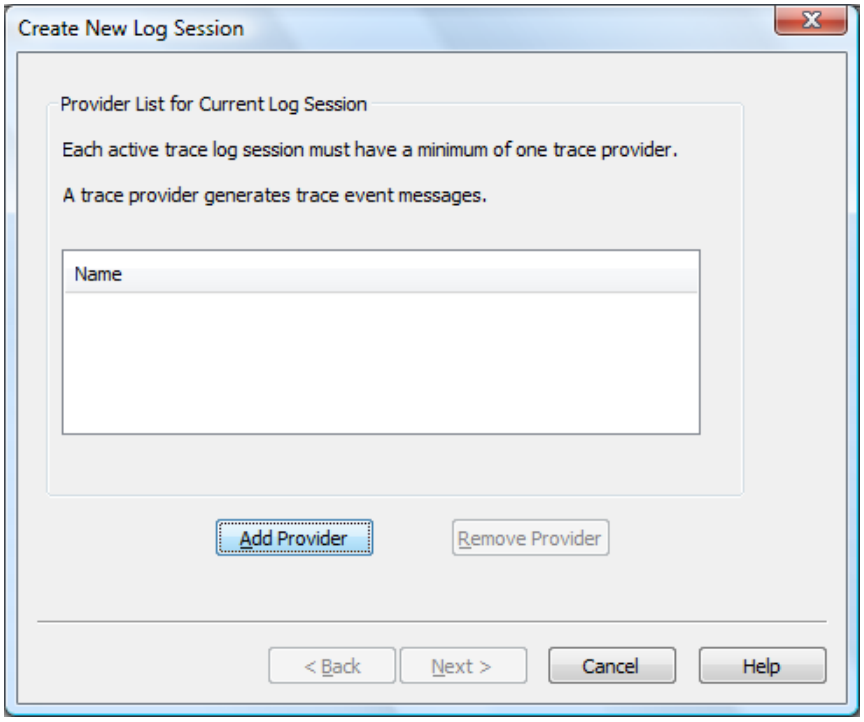
When we created our application, the project template, generated a unique identifier to be used as the application's ETW provider ID. We'll need to get that in order to tell TraceView what provider to listen for. To do that, merely open the .csproj file in your favorite text editor and copy the value of the EtwProviderId property:



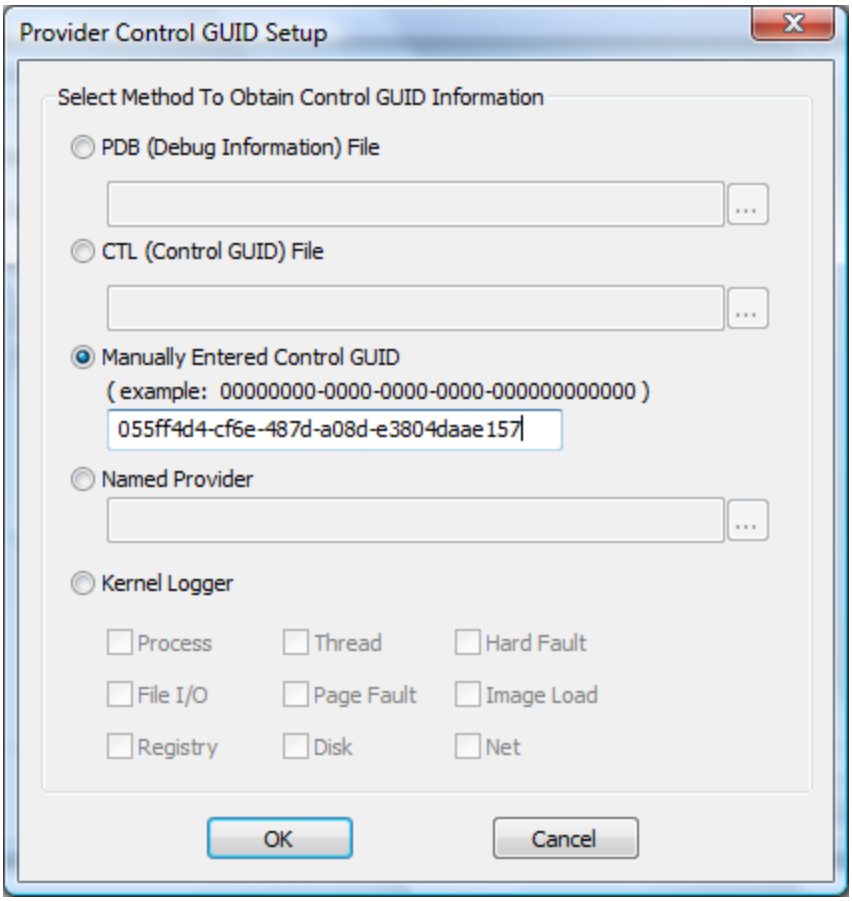
Then we can start TraceView and choose "Create New Trace Session" from the File Menu:



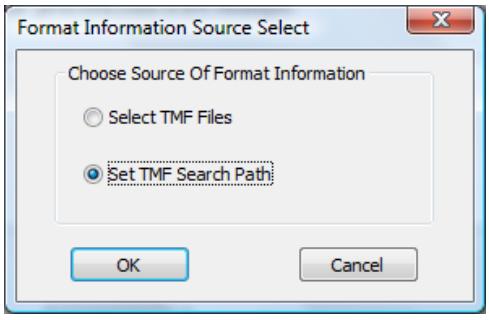
Once we've clicked that menu item, we will be presented with a "Create New Log Session" dialog. Now we need to add our provider ID to the list by clicking the "Add Provider" button.



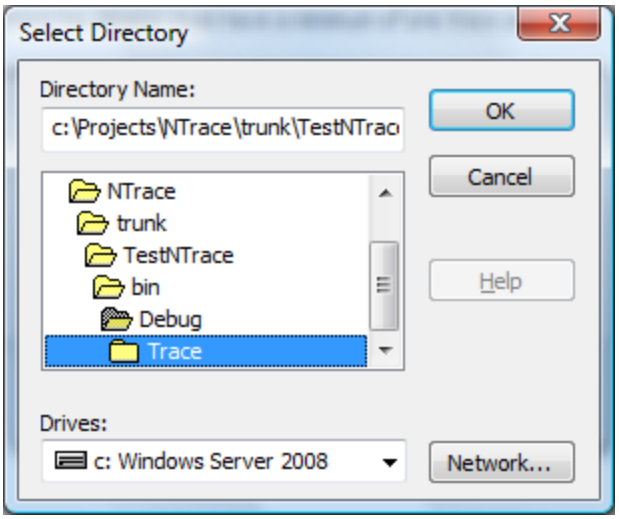
We are then presented with a dialog asking us what trace provider we'll be using. In this case, we have our provider ID:



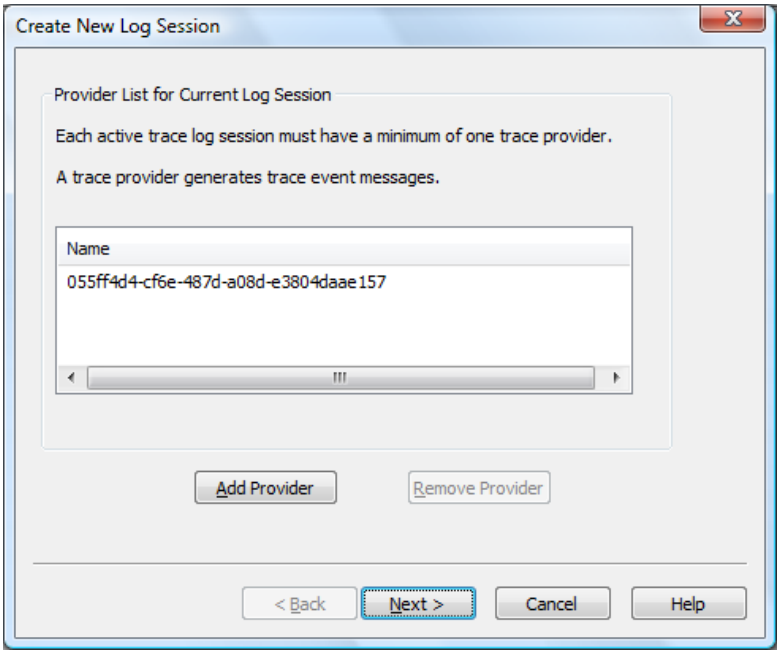
After we click the OK, button, we are then prompted for where TraceView will get the formatting information. In this case, we'll choose "Select TMF Files."



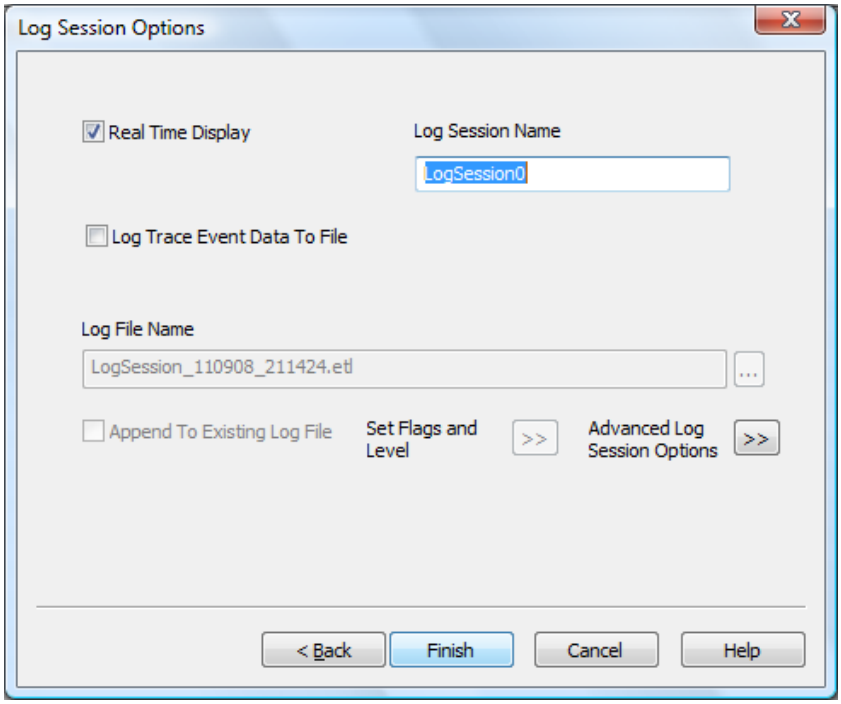
At this point, we are prompted with a dialog requesting the location TraceView should use to search for TMF files to use to generate the human-readable text. The NTrace preprocessor currently places the TMF and TMC files in the Trace folder for the current configuration's output folder. In this case, we'll use the files generated during the Debug build, so we'll navigate to the bin\Debug\Trace folder and click OK.



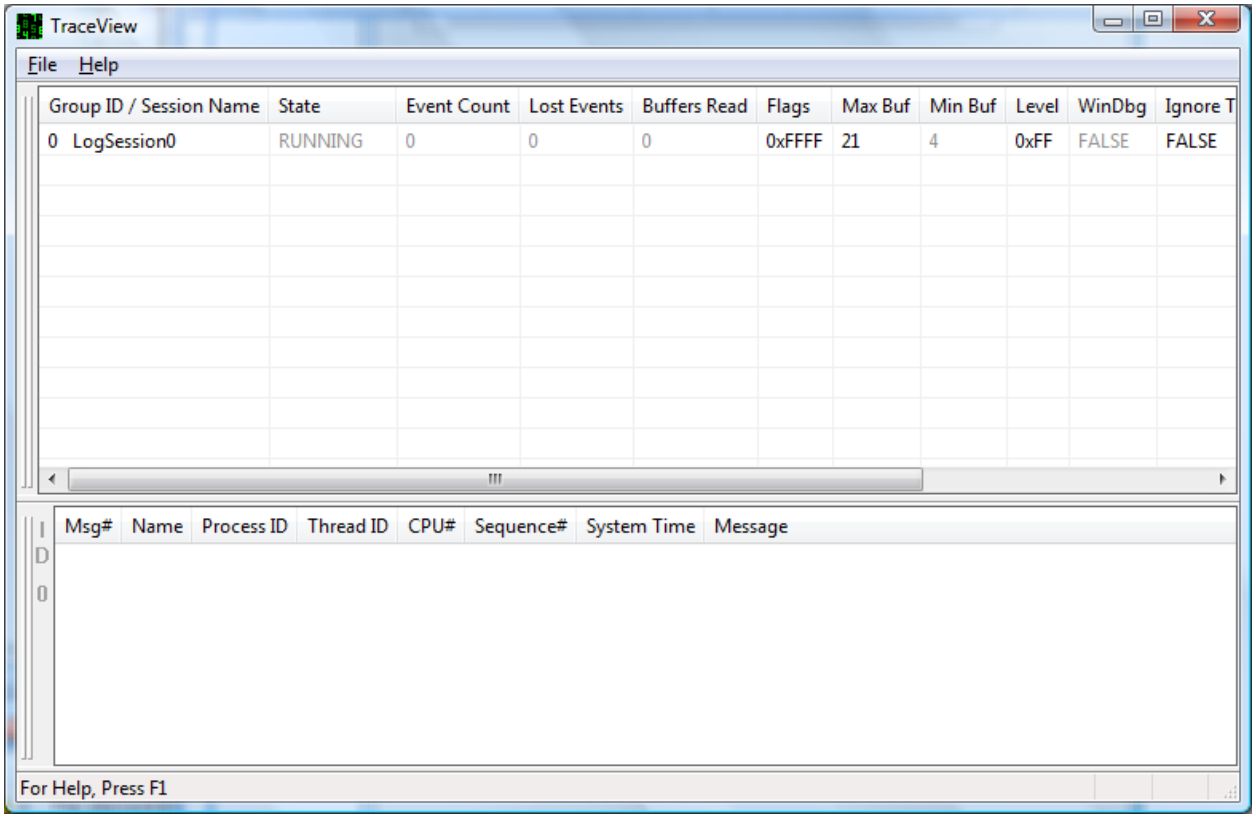
This returns us to the “Create New Log Session” dialog:



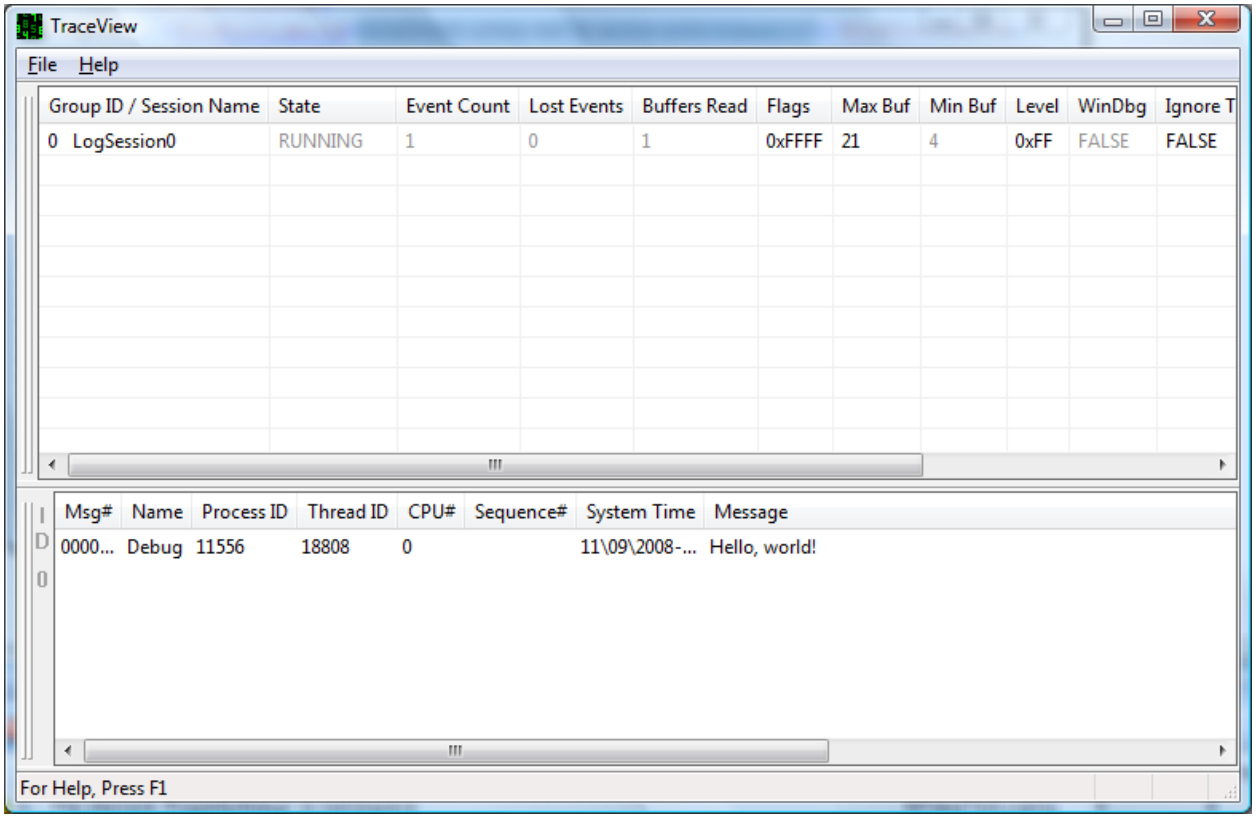
After clicking the Next button, we are presented with the final step: we need to choose whether we’ll view the trace session in real time, or if we will log to a file. In this case, we want to see the log messages in real time, so we’ll ensure that checkbox is checked:



There! We've created our log session and it is now waiting for trace messages to arrive:



We can now run our application and see trace messages appear in the console:


TraceView
File Help
Group ID / Session Name State Event Count Lost Events Buffers Read Flags Max Buf Min Buf Level WinDbg Ignore T
0 LogSession0 RUNNING 1 0 1 0xFFFF 21 4 0xFF FALSE FALSE
Msg# Name Process ID Thread ID CPU# Sequence# System Time Message
0000... Debug 11556 18808 0 11\09\2008-... Hello, world!
For Help, Press F1

Appendix A: An ETW performance demonstration

```
namespace TestMWPP
{
  using System;
  using System.Diagnostics;

  static class Program
  {
    internal static NTrace.ClassicProvider tracer;
    private static TraceSource traceSource =
        new TraceSource("Program");

    static int Main(String[] args)
    {
      const int NumIterations = 1000000;
      int dummy = 0;
      DateTime start, stop;
      start = DateTime.Now;
      for (int index = 0; index < NumIterations; index++)
      {
        dummy = dummy / (index + 1);
      }
      stop = DateTime.Now;
      Console.WriteLine("No Tracing: {0} milliseconds.", (stop -
start).TotalMilliseconds);

      start = DateTime.Now;
      for (int index = 0; index < NumIterations; index++)
      {
        dummy = dummy / (index + 1);
        traceSource.TraceEvent(TraceEventType.Information, 1, "Test " +
index.ToString());
      }
      stop = DateTime.Now;
      Console.WriteLine(".NET Tracing: {0} milliseconds.", (stop -
start).TotalMilliseconds);

      start = DateTime.Now;
      for (int index = 0; index < NumIterations; index++)
      {
        dummy = dummy / (index + 1);
        ETWTrace.Trace(EtwTraceLevel.Information, EtwTraceFlag.Component,
"Test {0}", index);
      }
      stop = DateTime.Now;
      Console.WriteLine("WPP Tracing: {0} milliseconds.", (stop -
start).TotalMilliseconds);

      String bar = "hi";
      int foo = 1;
      EtwTrace.Trace(EtwTraceLevel.Information, EtwTraceFlag.Diagnostic, "Hi
there! This is a {0} test.{1}", foo, bar);
```

```
      EtwTrace.Trace(EtwTraceLevel.Warning, EtwTraceFlag.Component, "Howdy!
This is another {0} test.{1}", bar, foo);
      EtwTrace.Trace(EtwTraceFlag.Component, "Hi! This is another {0}
test.{1}", bar, foo);
      return 0;
    }
  }
}
```