Using PLA.dll to collect perf counters

Who doesn’t love diagnostics?! I am a big proponent of **efficient** and **well thought out** [tracing](http://msdn.microsoft.com/en-us/magazine/cc163437.aspx) and [performance counters](http://msdn.microsoft.com/en-us/library/windows/desktop/aa371643(v=vs.85).aspx). They are invaluable for debugging, performance testing, health monitoring, and many other tasks worthy of future blog posts.

Seasoned Windows professionals are generally familiar with the Windows [Resource and Performance Monitor](http://technet.microsoft.com/en-us/magazine/2008.08.pulse.aspx?pr=blog) tool. Slightly more advanced users are typically aware of the [logman.exe](http://technet.microsoft.com/en-us/library/bb490956.aspx) tool which allows access to data collectors and performance logs via the command line. But in my experience, only a select few know about [PLA.dll](http://msdn.microsoft.com/en-us/library/windows/desktop/aa372631(v=vs.85).aspx), the programmatic interface to performance logs and alerts on Windows.

As it turns out, due to strong [COM interop support in .NET](http://msdn.microsoft.com/en-us/library/z6tx9dw3(v=vs.110).aspx) and the simplicity of adding COM reference assemblies in Visual Studio, it is a breeze to take advantage of PLA in your C# apps. Just go to your project, open the “Add Reference…” dialog, select “Browse…”, locate “pla.dll” (ships with Windows, typically in the %systemroot%\system32 folder), add it, and you’re ready to go.

Since PLA.dll is a COM library and geared mostly towards C++ developers, it is tricky to get the hang of using it in managed app. This is why I like to wrap it in a simpler, .NET-friendly façade when I’m exposing it to larger applications. (Pro tip: using the [protocol documentation for MS-PLA](http://msdn.microsoft.com/en-us/library/cc238489.aspx) can fill in some details left out by the MSDN documentation.)

Here is a sample wrapper that shows one way to expose performance counter collection via [data collector sets](http://technet.microsoft.com/en-us/library/cc749337.aspx). All of this code is available on [the PlaSample project in GitHub](https://github.com/brian-dot-net/writeasync/tree/master/projects/PlaSample).

We start with a type representing a counter name:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class CounterName  {      public CounterName()      {      }        public string Machine { get; set; }        public string Category { get; set; }        public string Counter { get; set; }        public string Instance { get; set; }        public override string ToString()      {          // . . .      }  } |

Now we abstract the data collector set into a CounterCollectorInfo:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class CounterCollectorInfo  {      public CounterCollectorInfo(string name)      {          this.Name = name;          this.CounterNames = new List<CounterName>();      }        public string Name { get; private set; }        public string OutputPath { get; set; }        public TimeSpan? SampleInterval { get; set; }        public LogFileFormat? LogFileFormat { get; set; }        public IList<CounterName> CounterNames { get; private set; }  } |

The LogFileFormat enum is mirror of the underlying PLA enum describing, predictably, the format of a perf counter log file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public enum LogFileFormat  {      CommaSeparated = 0,      TabSeparated = 1,      Sql = 2,      Binary = 3,  } |

Now for the nitty-gritty of interacting with PLA — code to create the data collector set for logging perf counters:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | public ICollectorSet Create()  {      // Data collector set is the core abstraction for collecting diagnostic data.      DataCollectorSet dcs = new DataCollectorSet();        // Set base folder to place output files.      dcs.RootPath = this.OutputPath;        // Create a data collector for perf counters.      IPerformanceCounterDataCollector dc = (IPerformanceCounterDataCollector)dcs.DataCollectors.CreateDataCollector(DataCollectorType.plaPerformanceCounter);      dc.name = this.Name + "\_DC";      dcs.DataCollectors.Add(dc);        // Set output file name to use a pattern, as described at      // <http://msdn.microsoft.com/en-us/library/windows/desktop/aa372131>(v=vs.85).aspx .      dc.FileName = this.Name;      dc.FileNameFormat = AutoPathFormat.plaPattern;      dc.FileNameFormatPattern = @"\-yyyyMMdd\-HHmmss";        // Set sample interval, if present.      if (this.SampleInterval.HasValue)      {          dc.SampleInterval = (uint)this.SampleInterval.Value.TotalSeconds;      }        // Set log file format, if present.      if (this.LogFileFormat.HasValue)      {          dc.LogFileFormat = (FileFormat)this.LogFileFormat.Value;      }        // Build up the list of performance counters.      string[] counterNames = new string[this.CounterNames.Count];      for (int i = 0; i < this.CounterNames.Count; ++i)      {          counterNames[i] = this.CounterNames[i].ToString();      }        dc.PerformanceCounters = counterNames;        // Now actually create (or modify existing) the set.      dcs.Commit(this.Name, null, CommitMode.plaCreateOrModify);        // Return an opaque wrapper with which the user can control the session.      return new CollectorSetWrapper(dcs);  } |

The interface ICollectorSet provides a simple non-PLA-dependent API for interacting with the data collector set:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public interface ICollectorSet : ISessionController  {      void Delete();  }    public interface ISessionController  {      void Start();        void Stop();  } |

Finally, pulling it all together, a sample application to create a basic counter set and output a CSV file. Note that manipulating data collector sets [requires special privileges](http://technet.microsoft.com/en-us/library/cc722184.aspx). The simplest way to avoid “access denied” errors is to just run any PLA app elevated.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | CounterCollectorInfo info = new CounterCollectorInfo("MyCounters");    info.SampleInterval = TimeSpan.FromSeconds(1.0d);  info.LogFileFormat = LogFileFormat.CommaSeparated;  info.OutputPath = Environment.CurrentDirectory;    info.CounterNames.Add(new CounterName() { Category = "Process", Counter = "Thread Count", Instance = "explorer" });  info.CounterNames.Add(new CounterName() { Category = "System", Counter = "System Calls/sec" });  info.CounterNames.Add(new CounterName() { Category = "Processor", Counter = "Interrupts/sec", Instance = "\_Total" });    ICollectorSet collector = info.Create();  collector.Start();    Thread.Sleep(5000);    collector.Stop();    collector.Delete(); |

Run the program and you’ll get an output CSV file named “MyCounters-” followed by a timestamp. The contents will look something like this:  
"(PDH-CSV 4.0) (Pacific Standard Time)(480)","\\Your-PC-Name\Process(explorer)\Thread Count","\\Your-PC-Name\System\System Calls/sec","\\Your-PC-Name\Processor(\_Total)\Interrupts/sec"  
"12/27/2013 12:24:33.231","39","46015.248698065603","3062.4673653801251"  
. . .

Using PLA.dll to collect ETW traces

As demonstrated previously, PLA.dll allows you to collect [perf counter logs](http://writeasync.net/?p=711). It can also be used to collect [ETW traces](http://msdn.microsoft.com/en-us/magazine/cc163437.aspx).

To collect traces, you need to add one or more trace data providers to your collector. Trace data providers are identified by a GUID known as the ETW provider ID. Windows comes with many built-in providers. In addition, third-party services and applications will often register their own providers on installation. To get a list of provider names and IDs currently registered on your system, open an elevated command prompt and run logman.exe query providers.

To represent a provider, I added a ProviderInfo class to the sample. (Remember, all this code is available on the [GitHub PlaSample project](https://github.com/brian-dot-net/writeasync/tree/master/projects/PlaSample).)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class ProviderInfo  {      public ProviderInfo(Guid id)      {          this.Id = id;      }        public Guid Id { get; private set; }        public uint? Level { get; set; }        public ulong? KeywordsAny { get; set; }        public ulong? KeywordsAll { get; set; }  } |

The level indicates the highest trace level to collect; ETW typically uses levels 1-5 (critical, error, warning, informational, verbose). The keywords are 64-bit masks which can be used as custom filters (e.g. [the CLR defines many keywords](http://msdn.microsoft.com/en-us/library/ff357720(v=vs.110).aspx) in its trace provider), where “any” means “match events with any of these bits set” and “all” means “*only* match events with all these bits.”

For the trace collector, I have exposed many options to control the log file size, whether to use a circular buffer (i.e. keep overwriting the log file with newer events once it reaches a max size), how big the event buffer should be, and so on. (It should be noted that many of these work with perf counter collectors as well, but were omitted for simplicity.)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class TraceCollectorInfo  {      public TraceCollectorInfo(string name)      {          this.Name = name;          this.Providers = new List<ProviderInfo>();      }        public string Name { get; private set; }        public string OutputPath { get; set; }        public uint? BufferSizeInKB { get; set; }        public bool? Circular { get; set; }        public TimeSpan? FlushTimer { get; set; }        public TimeSpan? MaxDuration { get; set; }        public uint? MaxSizeInMB { get; set; }        public uint? MaximumBuffers { get; set; }        public uint? MinimumBuffers { get; set; }        public bool? Segmented { get; set; }        public IList<ProviderInfo> Providers { get; private set; }  } |

The PLA code to create a trace collector is fairly similar to the perf counter collector code. The differences are mostly in the number of properties to set and the way the provider list is built up (note the helper methods to simplify the optional value processing):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | public ICollectorSet Create()  {      // Data collector set is the core abstraction for collecting diagnostic data.      DataCollectorSet dcs = new DataCollectorSet();        // Set base folder to place output files.      dcs.RootPath = this.OutputPath;        // Create a data collector for traces.      ITraceDataCollector dc = (ITraceDataCollector)dcs.DataCollectors.CreateDataCollector(DataCollectorType.plaTrace);      dc.name = this.Name + "\_DC";      dcs.DataCollectors.Add(dc);        // Set output file name to use a pattern, as described at      // <http://msdn.microsoft.com/en-us/library/windows/desktop/aa372131>(v=vs.85).aspx .      dc.FileName = this.Name;      dc.FileNameFormat = AutoPathFormat.plaPattern;      dc.FileNameFormatPattern = @"\-yyyyMMdd\-HHmmss";        // Set various values (if present)      SetValue(dc, this.BufferSizeInKB, (d, v) => d.BufferSize = v);      SetValue(dc, this.Circular, (d, v) => d.LogCircular = v);      SetValue(dc, this.FlushTimer, (d, v) => d.FlushTimer = (uint)v.TotalSeconds);      SetValue(dc, this.MaximumBuffers, (d, v) => d.MaximumBuffers = v);      SetValue(dc, this.MinimumBuffers, (d, v) => d.MinimumBuffers = v);      SetValue(dc, this.MinimumBuffers, (d, v) => d.MinimumBuffers = v);        SetValue(dcs, this.MaxDuration, (d, v) => d.SegmentMaxDuration = (uint)v.TotalSeconds);      SetValue(dcs, this.MaxSizeInMB, (d, v) => d.SegmentMaxSize = (uint)v);      SetValue(dcs, this.Segmented, (d, v) => d.Segment = v);        // Build up the list of providers.      foreach (ProviderInfo providerInfo in this.Providers)      {          TraceDataProvider provider = dc.TraceDataProviders.CreateTraceDataProvider();          dc.TraceDataProviders.Add(provider);            provider.Guid = providerInfo.Id;          AddValue(provider.KeywordsAll, providerInfo.KeywordsAll);          AddValue(provider.KeywordsAny, providerInfo.KeywordsAny);          AddValue(provider.Level, providerInfo.Level);      }        // Now actually create (or modify existing) the set.      dcs.Commit(this.Name, null, CommitMode.plaCreateOrModify);        // Return an opaque wrapper with which the user can control the session.      return new CollectorSetWrapper(dcs);  }    private static void SetValue<TClass, TValue>(TClass c, TValue? v, Action<TClass, TValue> setValue) where TValue : struct  {      if (v.HasValue)      {          setValue(c, v.Value);      }  }    private static void AddValue<TValue>(IValueMap map, TValue? v) where TValue : struct  {      if (v.HasValue)      {          map.Add(v.Value);      }  } |

Now some sample code to show how to use the trace collector. This example collects kernel process traces for about five seconds, creating new files after every one second. Remember to run this elevated:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | private static void CreateTraceCollector()  {      TraceCollectorInfo info = new TraceCollectorInfo("MyTraces");        info.BufferSizeInKB = 64;      info.Segmented = true;      info.MaxDuration = TimeSpan.FromSeconds(1.0d);      info.OutputPath = Environment.CurrentDirectory;        // Microsoft-Windows-Kernel-Process      Guid providerId = new Guid("{22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716}");        info.Providers.Add(new ProviderInfo(providerId) { Level = 5 });        ICollectorSet collector = info.Create();      collector.Start();        Thread.Sleep(5000);        collector.Stop();        collector.Delete();  } |

After the app finishes, you should see files like the following:  
MyTraces-20140101-125602.etl  
MyTraces-20140101-125603.etl  
...  
These files can be decoded by Windows [Event Viewer](http://technet.microsoft.com/en-us/library/cc766042.aspx) or with tools like [tracerpt.exe](http://technet.microsoft.com/en-us/library/bb490959.aspx).

Using tracerpt.exe [file.etl] -o [file.xml], you can dump the traces to a human-readable XML file. The file will contain a sequence of <Event> elements such as the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <Event xmlns="<http://schemas.microsoft.com/win/2004/08/events/event>">    <System>      <Provider Name="Microsoft-Windows-Kernel-Process" Guid="{22fb2cd6-0e7b-422b-a0c7-2fad1fd0e716}" />      <EventID>8</EventID>      <!-- ... -->    </System>    <EventData>      <Data Name="ProcessID">    1788</Data>      <Data Name="ThreadID">  411224</Data>      <Data Name="OldPriority">16</Data>      <Data Name="NewPriority">10</Data>    </EventData>    <RenderingInfo Culture="en-US">      <Level>Information </Level>      <Opcode>Info </Opcode>      <Keywords>        <Keyword>WINEVENT\_KEYWORD\_CPU\_PRIORITY</Keyword>      </Keywords>      <Task>CpuPriorityChange</Task>      <Message>CPU priority of thread 411224 in process 1788 was changed from 16 to 10. </Message>      <Channel>Microsoft-Windows-Kernel-Process/Analytic</Channel>      <Provider>Microsoft-Windows-Kernel-Process </Provider>    </RenderingInfo>  </Event> |