The TraceEvent Library Programmers Guide

Vance Morrison

Contents

[Introduction: Strongly Typed (Semantic) Logging 3](#_Toc376497353)

[Basic Logging Architecture 3](#_Toc376497354)

[Multi-player 5](#_Toc376497355)

[Asynchronous 5](#_Toc376497356)

[Quick Start Example 5](#_Toc376497357)

[Component 1: the Event Provider (EventSource) 5](#_Toc376497358)

[Component 2: The Event Session (TraceEventSession) 7](#_Toc376497359)

[Component 3: The Event Processor (ETWTraceEventSource) 8](#_Toc376497360)

[Event Parsing ‘Magic’ (TraceEventParsers) 9](#_Toc376497361)

[Static vs. Dynamic TraceEventParsers 13](#_Toc376497362)

[Lifetime constraints on TraceEvent objects 14](#_Toc376497363)

[Review of the Fundamental TraceEvent Architecture 15](#_Toc376497364)

[What You Can Do With the TraceEvent Library 15](#_Toc376497365)

[ETW Limitations 17](#_Toc376497366)

[Next Step: Code Samples 18](#_Toc376497367)

[Diagnostic Techniques: Using PerfView 18](#_Toc376497368)

[Full dumps of ETW events in PerfVIew 18](#_Toc376497369)

[PerfView’s /KeepAllEvents feature 19](#_Toc376497370)

[Debugging EventSource Authoring 19](#_Toc376497371)

[Real Time Processing of Events 20](#_Toc376497372)

[Real Time Processing with IObservables 21](#_Toc376497373)

[Best Practices for Versioning ETW Events 22](#_Toc376497374)

[Higher level Processing: TraceLog 23](#_Toc376497375)

[Creating a TraceLog (an ETLX file). 24](#_Toc376497376)

[Using Call Stacks with the TraceEvent library 26](#_Toc376497377)

[The Ugly Details of Stack Resolution 27](#_Toc376497378)

[Source Code Support 28](#_Toc376497379)

[Building Compile-time TraceEventParsers using TraceParserGen 28](#_Toc376497380)

[Getting the XML Manifest 29](#_Toc376497381)

[Creating XML Manifests for EventSources 29](#_Toc376497382)

[Creating XML Manifests OS registered ETW providers 29](#_Toc376497383)

[Converting to XML Manifest to a TraceEventParser for a Provider 30](#_Toc376497384)

[Event Provider Discovery 30](#_Toc376497385)

[Publication vs. Registration 31](#_Toc376497386)

[On the fly filtering AND Logging using ETWReloggerTraceEventSource 32](#_Toc376497387)

# Introduction: Strongly Typed (Semantic) Logging

As long as there have been programs, developers have used logging system to diagnosis functional and performance problems. Logging systems break down into two broad categories:

1. Weakly typed: The logging system only knows how to log strings and thus all data is converted into strings before it is logged. Printf logging is an example of this. These are the simplest logging systems and are a fine choice for ad hoc investigations by humans, however they suffer if the volume of data increases or if there is a desire to manipulate the data being logged programmatically.
2. Strongly typed (also known as [Sematic Logging](http://blogs.msdn.com/b/agile/archive/2013/02/07/embracing-semantic-logging.aspx)): In these logging systems each event is assigned schema which defines
	1. The name of the event
	2. The names and types of all the data items that are logged along with the event
	3. Optional meta-data about the event (groups it belongs to, its verbosity etc.).

Because the events are strongly typed, the consumer of the logging data can assume that the events conform to the schema which makes processing the events considerably easier and less fragile. It also improves the speed and compactness of logging as well as the post processing (no need to print to strings and then parse back). The [Event Tracing for Windows](http://msdn.microsoft.com/en-us/library/windows/desktop/bb968803%28v%3Dvs.85%29.aspx) (ETW) system built into the Windows Operating System is an example of a strongly typed logging system (see [Introduction to ETW](http://msdn.microsoft.com/en-us/magazine/cc163437.aspx) for more)

In general strongly types logging systems make sense in the same places where strongly typed programming languages make sense, when

1. When the data is likely to be consumed by other automation rather than simply viewed by humans.
2. When the amount of data (scale of logging) is high and therefore puts a premium on logging efficiently.
3. When the logging system is intended to ‘last forever’, and will need to be constructed in a decentralized way by many diverse people over many versions.
4. When the code logging the data versions at a different rate than the code processing the event data.

Server scenarios (aka the ‘cloud’) generally value these attributes since they are likely to want to automate the monitoring task, deal with high scale, and be developed by many programmers over time. Thus strongly typed eventing is a natural fit for the cloud.

The TraceEvent library is [NuGET](https://www.nuget.org/) package that provides a set of .NET Runtime classes that make it easy to control and consume (parse) the strongly typed [Event Tracing for Windows](http://msdn.microsoft.com/en-us/library/windows/desktop/bb968803%28v%3Dvs.85%29.aspx) (ETW) events. It is intended to be used in conjunction with the [System.Diagnostics.Tracing.EventSource](http://msdn.microsoft.com/en-us/library/system.diagnostics.tracing.eventsource.aspx) class (which can generate ETW events) to form a complete, end-to-end, strongly typed logging system.

# Basic Logging Architecture

There are three basic parts to and ETW based logging system as shown in the figure below

1. The **Event Session** (represents the entity controlling the logging). The session has the ability to tell the providers of events to start and stop logging and control how verbose the logging is. It also has the ability to route the data to various places. It can indicate that the data should be directly written to a file (for maximum efficiency) or to send it to the session itself (for on the fly processing)
2. The **Event Provider** is the part of the logging system that is wired into the application to be monitored. Its job is to call a logging API when interesting things happen in the application.
3. The **Event Consumer** takes the data from the file or from the session and consumes it in some way, typically generating aggregate statistics and generating alerts.

Event

Manifest

(Schema)

Event

Data

Commands

File

TraceEventSource

TraceEventSession

EventSource

Event

Data

Corresponding to each of these ‘players’, the TraceEvent library has a class that supports that role.

1. The **Event Session** uses the **Microsoft.Diagnostics.Tracing.TraceEventSession** class.
2. The **Event Provider** uses the **Microsoft.Diagnostics.Tracing.EventSource** class.
3. The **Event Consumer** uses the **Microsoft.Diagnostics.Tracing.TraceEventSource** class.

It is not uncommon for each of these ‘roles’ to be implemented in a separate process, but it is also common for some of the roles to be combined (most commonly the Session and the Consumer are in the same process).

Because the logging system is strongly typed, each event conforms to a particular schema and the event consumer needs to know this schema to decode the event data. The ETW system calls this schema the [manifest](http://msdn.microsoft.com/en-us/library/windows/desktop/dd996930%28v%3Dvs.85%29.aspx) and one representation of this is an XML file, but it also is stored in a binary form for efficient processing. One of the responsibilities of an event provider is to publish a manifest for the events that it generates so that the event consumer can find it when it needs it. As we will see there are a number of different ways of doing this. This information flow is designated in the diagram above by the red arrow.

##  Multi-player

An important aspect of the architecture that is not obvious in the diagram above is that each of the elements of the diagram can have multiple instances. Thus there can be:

1. Multiple event providers, each emitting events for the part of the system they understand, and each of which has a manifest for the events it might emit.
2. Multiple event sessions each of which are gathering different sets of events for different purposes. Each of these sessions has the option of logging its events to a file or to the session itself in real time.
3. Multiple event consumers that can process the events from a file or session. It is not recommended however to have multiple consumers feeding from the same event session in real time. Instead it is simpler and better to have multiple session each feeding a unique event consumer in the real-time case.

## Asynchronous

Another fundamental property of the system that is not obvious from the diagram above is that the logging system is asynchronous. When a provider writes an event, it is a ‘fire and forget’ operation. The event quickly gets written to a buffer and the program continues. From this point on processing of the event is concurrent with the running of the application. This has a number of ramifications:

1. **Logging is fast and scalable**. Only the initial copy to the first buffer in the logging pipeline actually delays the program. Note that this is independent of the number of sessions or consumers. All the rest of the logging activity happens on other threads of execution and can be parallelized (thus it scales well).
2. **Logging has minimal and predictable impact on the program**. There are no ‘global locks’ that need to be taken by the provider when it logs events. Because of this it is much more likely that behavior of threads (races) with logging on will be essentially the same (statistically speaking) as with logging off.
3. **There is the possibility of lost events**. If the providers generate event faster than the file can store them (or the real time processor can process them), then eventually events need to be dropped. This is the price you pay for making the writing of an event be asynchronous. The system will detect that events are lost, but the event data itself is unrecoverable.

# Quick Start Example

Enough theory. Let’s see how this works in practice.

## Component 1: the Event Provider (EventSource)

We start by logging some events using System.Diagnostics.Tracing.EventSource. You and paste this code into a console application project to have a fully working example (compiled against V4.5 or later of the runtime) if you wish.

using System.Diagnostics.Tracing;

[EventSource(Name = "Microsoft-Demos-MySource")]

class Logger : EventSource

{

 public void MyFirstEvent(string MyName, int MyId) { WriteEvent(1, MyName, MyId); }

 public void MySecondEvent(int MyId) { WriteEvent(2, MyId); }

 public static Logger Log = new Logger();

}

class Program

{

 static void Main(string[] args)

 {

 Logger.Log.MyFirstEvent("Hi", 1);

 Logger.Log.MySecondEvent(1);

 }

}

Above is source code for a program that logs two events using EventSource class. To be strongly typed, we must provide a schema for each event, and we do this by defining a subclass of the ‘EventSource’ class. This class will have an instance method for each event that can be logged by this provider. In the example above we define two events.

1. The MyFirstEvent that logs a string ‘MyName’ and an integer ‘MyId’.
2. The MySecondEvent that logs just the integer ‘MyId’ (which allows it to be correlated with the corresponding MyFirstEvent.

Notice that the method definitions provide all the information needed to generate the schema for the events. The name of the method defines the name of the event, and the argument names and types provide the names and types of each of the properties associate with the event.

In a more perfect world, humans would only author the declarations of an EventSource class since it is these declarations that specify the programmer’s intent. However to make these methods actually log events, the user need to define a ‘boiler plate’ body for each event that does two things

1. Defines a numeric value associated with the event. This is the first parameter to the ‘WriteEvent’ method call and is used to identify the event in all further processing (the event name is only used to generate the manifest). These event numbers start at 1 (0 is reserved) and by default **needs to be the ordinal number of the method in the class**. Thus it would be an error to reverse the order of the ‘MyFirstEvent’ and ‘MySecondEvent’ declarations above without also changing the first parameter to ‘WriteEvent’ to match the order in the class. If this restriction bugs you we will see how to avoid it later, but it will mean more typing on your part.
2. Passes along all the arguments from the method to the WriteEvent method. Because the arguments to the event method are used to generate the manifest, and the manifest is supposed to accurately describe the event, it would be an error to more or fewer arguments to WriteEvent. Thus the WriteEvent method is intended to be used only in this very particular way illustrated above.

The Logger class also has an attribute that defines the name for this provider to be ‘Microsoft-Demos-MySource’. If this attribute had not been provided the name of the provider would have been the name of the class without any namespace (e.g. Logger). If your provider is for more than ad-hoc logging, it is **STRONGLY** encouraged that you define a ‘real’ name for it that avoids collisions and helps your users understand what information your provider will log. We should follow the ‘best practices’ which the Windows Operation system group uses by making our name:

* Start with the company name first (unique world-wide).
* Then follow it with the name of a product or family of products.
* Use further sub-groups as needed.
* All separated by dashes (-).

Finally, in our example above the Logger class also defines a global static variable which creates an instance of the class that we use to log events. Having more than one instance of a particular EventSource is not recommended, since there is a cost to construct them, and having two serves no useful purpose. Thus most EventSources will have a single instance, typically in a static variable that was auto-initialized as shown above.

Once we have our ‘Logger’ EventSource defined, we simply call the event methods to log events. At this point we have an application with a fully functional ETW provider. Of course those events are off by default, so this program does not do anything yet, which brings us to step 2.

## Component 2: The Event Session (TraceEventSession)

To turn on events we need an Event Session, which is defined by the TraceEventSession class. Typically this session will be in another process (typically some data-collection service or program but it can even by the process logging the event). Here is code that does that. (Again you can cut and paste this into a console application which has referenced the [TraceEvent Nuget Library](http://www.nuget.org/packages/Microsoft.Diagnostics.Tracing.TraceEvent) to have a complete program)

using Microsoft.Diagnostics.Tracing.Session;

class Program

{

 static void Main()

 {

 using (var session = new TraceEventSession("MySession", "MyEventData.etl"))

 {

 session.EnableProvider("Microsoft-Demos-MySource");

 System.Threading.Thread.Sleep(10000);

 }

 }

}

In the code above, the program:

1. Creates a new TraceEventSession, each session is given a name that is unique ACROSS THE MACHINE. In our case we called our session ‘MySession’. Sessions CAN live beyond the lifetime of process that created them, and this name is how you refer to these sessions from other processes besides the process that created them. We also specify a file where the data is to be logged. By convention, these data files use the .ETL (Event Trace Log) suffix.
2. Once we have a session, we need to enable the providers. This can be done either by specifying the name of the provider or by using the unique 8 byte GUID that was assigned to the provider. Typically you will do this by name, which is what we do here, but in future example we will see cases where using the GUID is more convenient.
3. Waits for events to come in. In this case we simply wait 10 seconds. During this time any process that has ‘Logger’ instances in it will log methods to this file. Both existing and newly created processes will log events.
4. Next we close the session. TraceEventSession implement [IDisposable](http://msdn.microsoft.com/en-us/library/yh598w02.aspx) and the ‘using’ clause will naturally dispose the session when the ‘session’ variable goes out of scope. This causes the session to be torn down and the MyEventData.etl file will be closed.

That is it, thus running the program and then running the previous program simultaneously, will create a MyEventData.etl file with two events in it (the MyFirstEvent and MySecondEvent).

## Component 3: The Event Processor (ETWTraceEventSource)

Now that we have a data file, we can process the data. This is what the ETWTraceEventSource class does.

using Microsoft.Diagnostics.Tracing;

using System;

class Program

{

 static void Main()

 {

 using (var source = new ETWTraceEventSource("MyEventData.etl"))

 {

 // Set up the callbacks

 source.Dynamic.All += delegate(TraceEvent data) {

 Console.WriteLine("GOT EVENT {0}", data);

 };

 source.Process(); // Invoke callbacks for events in the source

 }

 }

}

In the program above we

1. Create an ETWTraceEventSource that uses the ‘MyEventData.etl’ file as its data source. The ETWTraceEventSource represents the stream of events as a whole. Because this class needs to support real time processing of events it does not use the ‘IEnumerable’ (pull model) for processing the events but rather provides a way of registering a callbacks.
2. Register callback for any events we are interested in. In this case we register a delegate that receives ‘data’ (of type ‘TraceEvent’) and prints ‘GOT EVENT’ and the event data for ‘All’ events in the ‘Dynamic’ group associated with the session. (More on this mysterious ‘Dynamic’ property shortly).
3. Call the ‘Process’ method, which causes the callbacks to actually be called. This method only returns when there are no more events (end of file), or because processing was explicitly stopped prematurely.
4. We dispose of the ETWTraceEventSource (the using clause does this for us when ‘source’ goes out of scope). This closes the file and release all resources associated with the processing.

Running the program above will result in the following output:

GOT EVENT: <Event MSec= "12.8213" PID="4612" TID="12372" EventName="MyFirstEvent" ProviderName="Microsoft-Demos-MySource"

 MyName="Hi" MyId="1"/>

GOT EVENT: <Event MSec= "13.7384" PID="4612" TID="12372" EventName="MySecondEvent" ProviderName="Microsoft-Demos-MySource"

 MyId="1"/>

Some useful things to call out about the output:

* The event is automatically stamped with a very accurate timestamp (typical resolution is 10 nsec or so) which indicates exactly when in time the event occurred.
* The process and thread on which the event fired is also captured automatically.
* The name of the provider as well as the event name are also captured.
* All data items in the payload are also decoded. The names of the data items ‘MyName’ and ‘MyId’ are recorded (as well as their types) along with the specific value.

In this example we simply use the ‘ToString’ method to print an XML representation of the event, however there are APIs for getting at all the data items above (including the payload values) in a convenient programmatic way. This is the real ‘value add’ of strongly typed logging.

At this point we have constructed an end-to-end example, creating a controller (TraceEventSession) that activated a ETW provider (our Logger EventSource) and send the data to a file which we then read with a consumer (ETWTraceEventSource) to pretty print the resulting events.

# Event Parsing ‘Magic’ (TraceEventParsers)

We have so far glossed over exactly how the ‘data.ToString()’ call was able to determine the names of the events (e.g. MyFirstEvent) as well as the names of the arguments (e.g. MyName and MyId) of the two events that were logged. We explain this important aspect here.

The ETW architecture breaks event data needed by parsers into two categories:

1. Information which is known before an event is actually fired (e.g. provider names, event names, arguments, types, verbosities …). This schema data can be described by XML called a manifest. Each event provider conceptually has a manifest that describes the events it might log.
2. Data that is only known at logging time and needs to be serialized into the data stream. (E.g. the string and integer values passed to MyFirstEvent and MySecondEvent, but NOT the names ‘MyFirstEvent’ and ‘MySecondEvent’ (they are in the first category)).

An important architectural point is that an ETWTraceEventSource is a source of UNPARSED events. These are resented by the Microsoft.Diagnostics.Tracing.TraceEvent class. These unparsed events will only know what is known **without looking at a manifest**. Thus it knows things like the timestamp, process ID, provider ID (GUID) and event ID (the small integer you pass to EventSource.WriteEvent). It even has the payload blob of bytes, but it does not know how to interpret them.

This is why although you CAN subscribe to events directly from EtwTraceEventSource, (this is what the ‘AllEvents’ registration point does), generally you don’t want to do this because the event you get back will not be able to give you names of events or decode the payload values.

Instead you need to hook the EtwTraceEventSource up to a TraceEventParser. As its name implies, a TraceEventParser is a class that knows how to parse some set of events. For example there is a class called KernelTraceEventParser that knows how to parse most of the events that can be generated by the Windows OS kernel. When this parser is created, it is ‘attached’ to a particular TraceEventSource. It in turn exposes a set of callback registration points (events in the C# sense) that allow you to subscribe to particular **parsed** events. For example the KerelEventParser has the subscription point:

 public event Action<ProcessTraceData> ProcessStart;

which fires whenever a new process starts on the machine. You can subscribe to this event like so

 kernelParser.ProcessStart += (ProcessTraceData data) => Console.WriteLine(data.CommandLine);

Which creates a C# delegate (the => operator) which takes ProcessTraceData, and prints out the command line that started the process. Thus a parser

1. Has a subscription point (a C# event) for every event it knows about (in this case we see the ‘ProcessStart’ event.
2. For each such event, if there are payload values, then there is a specific subclass of TraceEvent, which defines properties for each data item in that event. In the example above the ProcessStart event has payload values (event arguments) like ‘CommandLine’ and ‘ParentID’ and you can simply use C# property syntax to get their value, and the properties have the types you would expect them to have (string for ‘CommandLine’ and integer for ‘ParentID’).

Here is where the ‘strong typing’ of the logging becomes apparent. On the one end an EventSource logs strongly typed set of arguments (with names), and on the other end they pop out of a TraceEventParser as a class that has properties that let you fetch the values with full type fidelity.

Thus in general you get a diagram that looks like this:

Unparsed

Events

Unparsed

Events

Parsed

Events

Parsed

Events

Where potentially many different TraceEventParsers are ‘attached’ to a TraceEventSource and then in turn many callbacks are registered to the parser that knows how to decode them. The result is that you get fully parsed events in your callback code. Here is code that shows how to ‘connect’ the KernelTraceEventParser class to an ETWTraceEventSource and then subscribe to a the ‘ProcessStart’ event and fetch out the process name and command line. Notice that in the callback delegate we specify a specific subclass of TraceEvent called ‘ProcessTraceData’ which in addition to all the generic properties of an event (Name, Process, Timestamp …) also has properties for those specific to the ProcessStart event (e.g. ProcessName, CommandLine, ProcessID …).

using Microsoft.Diagnostics.Tracing;

using Microsoft.Diagnostics.Tracing.Parsers;

using Microsoft.Diagnostics.Tracing.Parsers.Kernel;

using System;

class Program {

 static void Main() {

 // Get a source that can return the raw events.

 using (var source = new ETWTraceEventSource("MyEventData.etl")) {

 // Connect a parser that understands kernel events

 var kernelParser = new KernelTraceEventParser(source);

 // Subscribe to a particular Kernel event

 kernelParser.ProcessStart += delegate(ProcessTraceData data) {

 Console.WriteLine("Process {0} Command Line {1}",

 data.ProcessName, data.CommandLine);

 };

 source.Process(); // call the callbacks for each event

 }

 }

}

Certain parsers (the Kernel, Clr and Dynamic parsers) are so common that there is a shortcut property that makes it easier than the code above. In the code below we use the ‘Kernel’ property on the ETWTraceEventSource object to get the kernel parser, and so the code can be simplified to the following.

using Microsoft.Diagnostics.Tracing;

using Microsoft.Diagnostics.Tracing.Parsers.Kernel;

using System;

class Program {

 static void Main() {

 // Get a source that can return the raw events.

 using (var source = new ETWTraceEventSource("MyEventData.etl")) {

 // Set up callback for a particular event

 source.Kernel.ProcessStart += delegate(ProcessTraceData data) {

 Console.WriteLine("Process {0} Command Line {1}",

 data.ProcessName, data.CommandLine);

 };

 source.Process(); // call the callbacks for each event

 }

 }

}

The TraceEvent library comes with a number of TraceEventParsers built in (in the namespace Microsoft.Diagnostics.Tracing.Parsers) including

* KernelTraceEventParser – which knows about windows OS kernel events. These include DLL loading, process start, stop, CPU sampling, page faults, Disk I/O file I/O, memory….
* ClrTraceEventParser – which knows about Common Languages Runtime (.NET CLR) events. These include GC events, Just in Time compilation events, Exception events …
* DynamicTraceEventParser – which knows about any event provider that have ‘Dynamic’ manifests, in the sense that manifest information is dumps into the event stream using a stand convention. All EventSources follow this convention, and thus all EventSources can be parsed by this parser.
* RegisteredTraceEventParser – which knows about any event provider that registers itself with the operating system (using the wevtutil command)). This includes most providers that ship with the windows operating system that are NOT the kernel provider or EventSources. You can see a list of such providers with the ‘logman query providers’ command.
* WPPTraceEventParser – which knows how to parse events written with the [WPP Tracing](http://msdn.microsoft.com/en-us/library/windows/hardware/ff556204%28v%3Dvs.85%29.aspx) system. Device drivers and other low level components often use this mechanism.
* JScriptTraceEventParser – which knows about the JavaScript runtime events
* TPLTraceEventParser – which knows about the Thread Parallel Library (another name for classes in the System.Threading.Tasks namespace).
* ASPNetTraceEventParser – which knows about ASP.NET events.

As mentioned the first three providers are so common, that the ETWTraceEventSource has three shortcut properties (Kernel, CLR and Dynamic), that allow you to access these providers in a very easy way.

We are now **finally** in a position to explain the mysterious piece of code in our original example that parsed EventSourceEvents:

 source.Dynamic.All += delegate(TraceEvent data) { … }

This code takes the EtwTraceEventSource ‘source’ and fetches the DynamicTraceEventParser using the ‘Dynamic’ shortcut, and then asks that parser to call it back on all events THAT THAT PARSER KNOWS ABOUT. Thus All does not mean every event coming in from the source, but just all events that a DynamicTraceEventPaser can parse (which is basically any event generated from an EventSource).

## Static vs. Dynamic TraceEventParsers

You will notice that in our first example where we were parsing EventSource events using the DynamicTraceEventParser the callback was declared as receiving its event data as a TraceEvent, and not some subclass (like ProcessTraceData). This might lead you to believe that the data is not parsed, and you would be half-right. Because parsers like DynamicTraceEventParser only learn about the event schema at runtime, it does not even ‘make senses’ to generate specific types like ProcessTraceData that have the right properties because we simply don’t know these at compile time. However EventSources DO log this manifest information to the ETW stream itself, so the information needed to decode the ETL file is in the file at runtime. Thus the DynamicTraceEventParser CAN decode the event. In particular there are functions like ‘TraceEvent.PayloadByName(string)’ which given a string name will return the payload for that property. For example here is the first example where we fetch the ‘MyName’ and ‘MyId’ fields from the ‘FirstEvent event. At compile time we only know that the return event is a ‘TraceEvent’ but the DynamicTraceEventParser is able to parse it can return type-correct values from the PayloadByName method.

using Microsoft.Diagnostics.Tracing;

using System;

class Program

{

 static void Main()

 {

 using (var source = new ETWTraceEventSource("MyEventData.etl"))

 {

 // Set up the callbacks

 source.Dynamic.AddCallbackForEvent("MyFirstEvent", delegate(TraceEvent data) {

 Console.WriteLine("GOT MyFirstEvent MyName={0} MyId={1}",

 data.PayloadByName("MyName"), data.PayloadByName("MyId"));

 });

 source.Process(); // Invoke callbacks for events in the source

 }

 }

}

Clearly this experience is a step down from what you get with a compile time solution. Certainly it is clunkier to write, and also error prone (if you misspell ‘MyName’ above the compiler will not catch it like it would if we misspelled ‘CommanLine’ when accessing process data). It is also MUCH less efficient to use ‘PayloadByName’ than to use compile time trace parser properties. What we would have LIKED to write is something like the following

 source.Dynamic.AddCallbackForEvent("MyFirstEvent", delegate(MyFirstEventTraceData data) {

 Console.WriteLine("GOT MyFirstEvent MyName={0} MyId={1}", data.MyName, data.MyId); });

Where we have an event-specific type that with a ‘MyName’ and ‘MyId’ property. Thus even though DynamicTraceEventParser is sufficient to parse events from an EventSource with full fidelity, if you are doing more than just printing the event, it is a good idea to create a static (compile time) parser that is tailored for your EventSource and thus can return compile time types tailored for your event payloads. This is what the TraceParserGen tool is designed to do, which we will cover later.

# Lifetime constraints on TraceEvent objects

In the most common scenarios involving the TraceEvent library, a user program will scan over a very large stream of events in an ETL file will only care about a small fraction of the data collected. Because of this, TraceEvent is tuned to be as efficient at scanning events as possible. To achieve this TraceEvent **AGGRESIVELY REUSES** the System.Diagnostics.Tracining.TraceEvent objects that it passed to user code.

Consider the following simple example that scans the ‘MyEventData.etl’ file for ‘ProcessStart’ events from the kernel to fine the first process that started during data collection.

using Microsoft.Diagnostics.Tracing;

using Microsoft.Diagnostics.Tracing.Parsers.Kernel;

using System;

class Program {

 static void Main() {

 ProcessTraceData firstProcess = null;

 using (var source = new ETWTraceEventSource("MyEventData.etl")) {

 source.Kernel.ProcessStart += delegate(ProcessTraceData data) {

 Console.WriteLine("Process {0} Command Line {1}",

 data.ProcessName, data.CommandLine);

 if (firstProcess == null) {

 // Subtle Error without Clone()

 firstProcess = (ProcessTraceData)data.Clone();

 }

 };

 source.Process();

 Console.WriteLine("The first process was {0} Command Line {1}",

 firstProcess.ProcessName);

 }

 }

}

The user registers a callback delegate with the ‘ProcessStart’ events associated with the KernelTraceEventParser. This delegate will be called with a ‘ProcessTraceEvent’ argument which represents the event. It is important to realize that **user code cannot keep references to this object after the callback has returned**. This is because the next ProcessStart event the library returns will likely overwrite this object with new data, leading to errors.

In the code above the user code wished to remember the data associated with a particular event (the first process start event), and thus has a problem. There are two options

1. Create a new user-defined object and copy out the necessary data into this object at scan time. Thus no reference to the TraceEvent object survives the callback.
2. Use the TraceEvent.Clone() method. This method will make a copy of TraceEvent object which will not be reused by the library.

As you can see, the code above opted for the second technique. All that is required is to call the Clone() method (and probably cast it back to its most specific type), before storing it in a long-lived reference. This is a moderately expensive operation (copying 50 or so bytes of data), so if you only need a handful of fields and you have an obvious user-defined structure on which to keep them, the first option (copying what you need out of the event), is more efficient.

While cloning events a bit of a pain, by aggressively reusing event object, in the common case you can scan a file with millions of events and not have to allocate any objects in the main scanning loop. This is an important performance win.

# Review of the Fundamental TraceEvent Architecture

At this point you have the fundamentals of the TraceEvent library.

1. TraceEventSession starts new ETW sessions and turn on ETW providers and direct the output.
2. ETW Providers you can turn on include:
	1. The Window OS Kernel Provider, .NET and Jscript Runtime, as well as most OS components.
	2. Any System.Diagnostics.Tracing.EventSource you create in your own apps.
3. You process the events by first hooking up a ETWTraceEventSource to the event stream (either a file or real time)
4. The events in the ETWTraceEventSource are unparsed. To parse them you need to hook the appropriate TraceEventParser up to your ETWTraceEventSource. These Parsers are either
	1. Static (Compile time) which are easier to use and very efficient but require that you know the schema of the events you wish to use at compile time. Static parsers include most of the ones that come with the TraceEvent library as well as ones you generate from manifest files using the TraceParserGen tool (discussed later).
	2. Dynamic (Run time) which are parsers that can process manifest information on the fly. There are only a handful of these including DynamicTraceEventParser, RegisteredTraceEventParser and WPPTraceEventParser. These are not as convenient to code against, and are less efficient, but are often used when the processing to be done is minimal (e.g. printing).
5. Once you have a parser, you subscribe to the C# events you are interested in and your callbacks will get strongly typed versions of the events that understand the payload.
6. Once you have hooked up your subscriptions, you call ETWTraceEventSource.Process() to start processing the ETW stream and calling the appropriate callbacks.

# What You Can Do With the TraceEvent Library

To keep things concrete we ‘went deep’ so far, showing real code, but by necessity we restricted ourselves to the simplest possible scenario using the TraceEvent library. Now we ‘go broad’ and describe the breath of the library so you can determine if the library is capable of handling the scenario you have in mind.

Capabilities include:

* The ability to monitor ETW events, sending them either to a file or directly to a programmatic callback in ‘real time’.
* The ability for those real time events to be passed to the [IObservable](http://msdn.microsoft.com/en-us/library/dd990377.aspx) interface and thus be used by the [Reactive Extensions](http://msdn.microsoft.com/en-us/data/gg577609.aspx).
* The ability turn on event providers selectively using ETW ‘Keywords’ and verbosity ‘Levels’. You can also pass additional arguments to your provider which EventSources can pick up. In that way you can create very sophisticated filtering specification as well as execute simple commands (e.g. force a GC, flush the working set, and etc.).
* The ability to enumerate the ETW providers on the system as well as in a particular process, and the ability to determine what ETW groups (Keywords) you can turn on.
* Ability to take ETL files and merge them together into a single file.
* Ability to read an ETL file or real time session and write an ETL file from it, filtering it or adding new events (Windows 8 only).
* The ability to capture stack traces when events are being logged.
* The ability to convert the stacks to symbolic form both for .NET, Jscript, as well as native code.
* The ability to store events in a new format (ETLX) that allows the event to be access efficiently in a random fashion as well as to enumerate the events backwards as well as forwards, and to efficiently represent the stack information.
* The ability to make generate C# code that implements a strongly typed parsers for any ETW provider with a manifest (TraceParserGen).
* The ability to read events written with the [WPP Tracing](http://msdn.microsoft.com/en-us/library/windows/hardware/ff556204%28v%3Dvs.85%29.aspx) system.
* The ability to access ‘Activity IDs’ that allow you to track causality across asynchronous operations (if all components emits the right events).
* Access Kernel events (along with stack traces), including:
	+ Process start/stop, Thread start/stop, DLL load and unload
	+ CPU Samples every MSec (but you can control the frequency down to .125 msec)
	+ Every context switch (which means you know where you spend blocked item) as well as the thread that unblocked the thread.
	+ Page faults.
	+ Virtual Memory allocation.
	+ C or C++ heap allocations.
	+ Disk I/O.
	+ File I/O (whether it hits the disk or not).
	+ Registry access.
	+ Network I/O.
	+ Every packet (with compete data) that comes on or off the network (network sniffer).
	+ Every System call.
	+ Sampling of Processor CPU counters (Instructions executed, branch mispredicts, Cache misses, …) (Windows 8 only).
	+ Remote Procedure calls.
	+ How the machine is configured (disk/ memory CPUs …).
* Access CLR (.NET Runtime) events, including:
	+ When GCs happen.
	+ When allocations are made (sampling and non-sampling).
	+ When object are moved during a GC.
	+ When methods are Just In Time (JIT) compiled.
	+ When Exceptions are thrown and the stack at which it was thrown.
	+ When System.Threading.Task.Tasks are created, and scheduled.
	+ Addition information on why a .NET assembly failed to load (to diagnose failures).
	+ Information to decode .NET frames in stack traces.
* Access ASP.NET events which log when request come in and when various stages of the pipeline complete.
* Access WCF events which log packets as the go through their pipeline.
* Access Jscript runtime events, including:
	+ Garbage collection.
	+ Just-in-Time (JIT) compilation of methods.
	+ Information to decode JScript frames in stack traces.

You can also get a reasonably good idea of what is possible by taking a look at the [PerfView](http://www.microsoft.com/en-us/download/details.aspx?id=28567) tool. PerfView was built on top of the TraceEvent library and all the ETW capabilities of that tool are surfaced in the TraceEvent library.

# ETW Limitations

Unfortunately, there are some limitations in ETW that sometimes block it from being used in scenarios where it would otherwise be a natural fit. They are listed here for emphasis.

* You send commands to providers on a machine wide basis. Thus you can’t target particular processes (however if you own the event provider code you can pass it extra information as arguments to ‘enable’ command to the provider and have your provider implement logic to ignore ‘enable’ commands not intended for it). (Fixed in Windows 8.1).
* Because commands are machine wide and thus give you access to all information on the system, you have to be Elevated (Admin) to turn an ETW session on or off.
* By design the communication between the controllers and the providers is ‘fire and forget’. Thus ETW is not intended to be a general purpose cross process communication mechanism. Don’t try to use it as such.
* In real time mode, events are buffered and there is at least a second or so delay (typically 3 sec) between the firing of the event and the reception by the session (to allow events to be delivered in efficient clumps of many events).
* Before Windows 8, there could only one kernel session. Thus using kernel mode events for ‘monitoring’ scenarios was problematic because any other tools that used kernel sessions were likely to interfere by overriding the single Kernel model event logging session.
* The file format is private, and before windows 8 could be quite space inefficient (it compresses 8-to-1). Files can get big fast.
* Logging more than 10K events /sec will load the system noticeably (5%). Logging more frequently than 10K/sec should be avoided if possible. Logging 1M events /sec will completely swamp a typical machine.

# Next Step: Code Samples

The best way to take the next step in learning about the TraceEvent library is to experiment with the [TraceEvent code samples](http://www.nuget.org/packages/Microsoft.Diagnostics.Tracing.TraceEvent.Samples). This is a [NuGet](http://www.nuget.org/packages) package that shows some simple but common scenarios. These samples are well commented with discussions on potential pitfalls and potential subtle design issues associated with the scenarios. They are worth the read. The Samples are in the TraceEvent Library Samples Package. A simple way of trying them out using Visual Studio is to:

1. Create a new Console program project.
2. Right click on the ‘References’ icon under the new project’s XXXX.\*Proj file. In solution explorer.
3. Select ‘Managed NuGet Pacakges’.
4. Search for ‘TraceEvent’ in the dialog that comes up.
5. Select ‘TraceEvent Library Samples’.

This will download the sample source code into your console application. The samples are in the ‘TraceEventSamples’ directory. Take a look at them. There is a README.TXT file to get you going, which will tell you to modify your program to call:

 TraceEventSamples.AllSamples.Run();

To Run all the samples. They should be self-explanatory.

# Diagnostic Techniques: Using PerfView

One prominent program that uses the TraceEvent library extensively is the [PerfVIew](http://www.microsoft.com/en-us/download/details.aspx?id=28567) tool. As its name suggests, this tools primary purpose is the collection, aggregation, and display of data useful for performance investigation, however it also is a valuable tool for simply enabling and controlling ETW event providers like EventSources as well as displaying ETW events in a human readable fashion.

In particular you can log the data from the ‘Microsoft-Demos-MySource’ EventSource we defined to a file with the PerfView command:

 PerfVIew collect /OnlyProviders=\*Microsoft-Demos-MySource

Please note the ‘\*’. This is PerfView’s way of indicating that the name should be converted to a GUID in the standard way that EventSources’s define. Once you have the data you can view it using the PerfView Event’s view. It is very easy to filter by process, time, event name or text in the events, as well as export the data to Excel or XML. The following PerfVIew videos are especially relevant.

* [Event Viewer Basics](http://channel9.msdn.com/Series/PerfView-Tutorial/Perfview-Tutorial-6-The-Event-Viewer-Basics)
* [Generating your own Events with EventSources](http://channel9.msdn.com/Series/PerfView-Tutorial/PerfView-Tutorial-8-Generating-Your-Own-Events-with-EventSources)

You will find having PerfVIew to be very handy when debugging your own ETW processing.

## Full dumps of ETW events in PerfVIew

Normally PerfView’s event view does not show all the data in an ETW file. Things like the Provider GUID, EventID, Opcode and payload bytes are not shown because they typically are not relevant. However when you are debugging ETW processing of your own, these low level details can be critical so you would like to be able to see them. You can do this selectively in PerfView by doing the following:

1. Find the Event that you are interested in (typically by looking at the timestamp).
2. Select the time cell of that event.
3. Right click and select -> Dump Event.

It will then open PerfView’s log that will contain a XML dump of the event, including all low level information (including the raw payload bytes).

## PerfView’s /KeepAllEvents feature

By default PerfView filters out some kernel and CLR events that it does not believe are of interest to you. These events happen in the beginning and end of the file (so called DC (data collection) start and stop events). Most of the time this is a good thing, but if there is a problem you are trying to debug with these events this filtering is a problem. It can be disabled by starting PerfView with the /keepAllEvents option. Also you may also have to use the File -> Clear Temp Files, GUI command if PerfView had already opened that particular ETL file. Otherwise it will simply use the old (filtered) data in a temp file.

## Debugging EventSource Authoring

While this document is really about TraceEvent, there is a good chance you will be authoring EventSources along the way. One pitfall of EventSources is that while they generate good diagnostic information when you make mistakes (like using the same event ID twice), these diagnostics only occur when the provider is actually enabled, and even then the exceptions that are thrown are swallowed by the runtime itself. The end effect is that the EventSource does not log events and it is not obvious why.

There are number of ways to diagnose error when authoring a new EventSource:

1. Develop the EventSource module using the [EventSource NuGet package](https://www.nuget.org/packages/Microsoft.Diagnostics.Tracing.EventSource). This package as a compile time rule that will look for errors in your EventSource and warn you about them at compile time. This produces the best experience.
2. You should do all your development with the EventSource turned on. You can do this with the following command:

PerfView /CircularMB=10 /OnlyProviders:\*Microsoft-Demos-MySource start

Which starts a circular logging session and leaves it on until you explicitly turn it off. Now you will get exceptions whenever your code runs, however they will still be swallowed. To fix that go to your Debug->Exceptions dialog and enable stopping on any thrown CLR exception. Any authoring mistakes will now be very obvious.

1. Use the ‘ConstructionException’ property. By default EventSources NEVER throw exceptions (so that turning on logging will not induce an error). However in DEBUG code you should have logic like this in your program that explicitly looks for errors during construction

 if (MySource.ConstructionException != null)

 throw MySource.ConstructionException;

This technique, along with (2) above (turning in your EventSource constantly during development) will insure that you find errors in your EventSource promptly error).

1. Look at the ‘Exceptions’ events in perfVIew when you attempt to use your EventSource (see [this blog entry](http://blogs.msdn.com/b/vancem/archive/2012/12/21/why-my-doesn-t-my-eventsource-produce-any-events.aspx) for more). Basically even though the EventSource does suppress exceptions during construction, they are actually occurring (but being swallowed). PerfView can see these exceptions and display them to debug the issue.

# Real Time Processing of Events

In the examples so far, we have seen how to collect events to a file, and read the events from a file. It is also possible to do this skipping the creating of the file altogether by using what are called ‘Real Time’ source. Here is what one looks like that watches in real time for processes to start.

using Microsoft.Diagnostics.Tracing.Parsers;

using Microsoft.Diagnostics.Tracing.Parsers.Kernel;

using Microsoft.Diagnostics.Tracing.Session;

using System;

class Program {

 static void Main() {

 // create a real time user mode session

 using (var session = new TraceEventSession("ObserveProcs")) {

 // Set up Ctrl-C to stop the session

 Console.CancelKeyPress +=

(object s, ConsoleCancelEventArgs args) => session.Stop();

 // Subscribe to a callback that prints the information we wish

 session.Source.Kernel.ProcessStart += delegate(ProcessTraceData data) {

 Console.WriteLine("Process {0} Command Line {1}",

 data.ProcessName, data.CommandLine);

 };

 // Turn on the process events (includes starts and stops).

 session.EnableKernelProvider(KernelTraceEventParser.Keywords.Process);

 session.Source.Process(); // Listen (forever) for events

 }

 }

}

You can see a few key differences between this code and the code that processes a file.

1. It combines both the collection and processing together, so you have both a TraceEventSession and a TraceEventSource.
2. When the TraceEventSession is created it takes only the session name, a filename parameter is not needed since there is no file generated.
3. We set up a ctrl-c handler to call ‘Stop()’ on the session when Ctrl-C is pressed. This stops what would otherwise be a infinite program.
4. You can get the TraceEventSource you need by accessing the ‘Source’ property on a TraceEventSession.
5. From here you subscribe to events in the same way (finding the desired parser and registering a callback.
6. Once you are set up, you need to turn on providers. In this case we turn on Kernel providers that generate events for Processes. Note that the Kernel provider is special and you need to call a special EnableKernelProvider API. User mode ETW provider (e.g. EventSources) would us the EnableProvider API as before. Here we are also showing the use of ‘keywords’ which are bitsets that indicate which events a provider CAN log should actually be logged. In our case we ask only for the Process events and not the many other events the Kernel can provide.
7. Once you are set up, you call Process() just before. For real time sessions this call will wait forever until another thread calls session.Stop() (which is what the Ctrl-C handler does).

When you run this program you will see that there is a noticeable delay of a few seconds between a process starting and the event callback being called. This is because ETW does not flush aggressively but waits a second or more before flushing. This is more efficient, but does mean a slight delay. Note that the events do have the very accurate timestamp that indicates exactly when the event occurred, so correlating events in time is still easy, there is just a delay before you get the data.

# Real Time Processing with IObservables

The [Linq](http://msdn.microsoft.com/en-us/library/bb397926.aspx) library is a set of data-base like operators for manipulating any collection of objects (more formally any [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0%28v%3Dvs.110%29.aspx)). It allows you to filter, map, select, group or sort the collection. However library does assume a ‘Pull model’ where the data object are passive, and the client is the master and ‘pulls’ the data from the collection one element at a time. The [Reactive Extensions](http://msdn.microsoft.com/en-us/data/gg577609.aspx) library is a .NET NuGet package that allows you do something very similar to Linq but for a ‘Push model’ stream. In the push model the client registers callback functions and the send data source is the master that ‘pushes’ the data one object at a time to the client on its schedule.

We have been using the push model in all the TraceEvent examples so far. It is characterized by having the user code register a callback and then having thread call a ‘Process()’ method to wait for incoming data. This model is forced upon you when you are doing real time monitoring, since the collection of event never ends since more events can come in at any point.

To support Linq-like query operators for ‘push style’ scenarios, we need something like IEnumerable<T> but set up for a push model. That is what [IObservable<T>](http://msdn.microsoft.com/en-us/library/dd990377%28v%3Dvs.110%29.aspx) is designed to do. The TraceEvent library exposes its streams of events as IObservable<T> objects, and this allows you to use the Linq-like query operators that the [Reactive Extensions](http://msdn.microsoft.com/en-us/data/gg577609.aspx) library defines to manipulate the data. It can be quite powerful.

Here is an example of using TraceEvent’s IObservable<T> support to print out a line to the console every time a process starts.

using Microsoft.Diagnostics.Tracing;

using Microsoft.Diagnostics.Tracing.Parsers;

using Microsoft.Diagnostics.Tracing.Parsers.Kernel;

using Microsoft.Diagnostics.Tracing.Session;

using System;

class Program {

 static void Main() {

 // create a real time user mode session

 using (var session = new TraceEventSession("ObserveProcs")) {

 // Set up Ctrl-C to stop the session

 Console.CancelKeyPress +=

 (object s, ConsoleCancelEventArgs args) => session.Stop();

 // Create a stream of process start events.

 var procStream = session.Source.Kernel.Observe<ProcessTraceData>("ProcessStart");

 // Subscribe to the stream, sending data to the console in real time.

 procStream.Subscribe(data =>

 Console.WriteLine("Process Started: Name {0} CmdLine {1}",

 data.ProcessName, data.CommandLine));

 // Turn on the process events (includes starts and stops).

 session.EnableKernelProvider(KernelTraceEventParser.Keywords.Process);

 session.Source.Process(); // Listen (forever) for events

 }

 }

}

As you can see this example is VERY similar to the previous example that used c# events to subscribe to the real time events. The only differences are

1. Instead of using the C# ProcessStart event to register the callback you use the Observe method on a TraceEventParser to create in IObservable<T> for specific events.
2. Once you have a IObservable<T>, you call its ‘Subscribe’ method to actually register the callback.

In the example above there is not a lot of value in using IObservable<T>. The value would come into play if you wish to use the [Reactive Extensions](http://msdn.microsoft.com/en-us/data/gg577609.aspx) Linq-like operators to form complex transformations on the events.

Note that UNLIKE the IEnumerable (TraceEvents) interface, the IObservable interfaces by default ALREADY do a Clone() of the TraceEventObject before returning it. Thus there is no need for you to call Clone() if the event came from an IObservable interface.

The [TraceEvent code samples](http://www.nuget.org/packages/Microsoft.Diagnostics.Tracing.TraceEvent.Samples) nuget package has several examples of using the IObservable support.

# Best Practices for Versioning ETW Events

This section is really more about EventSource (production) than TraceEvent (consumption), but is likely relevant to users of TraceEvent as well.

At some point you are going to have deployed EventSource that generate events and parsers built on TraceEvent for processing the events. Inevitably you will wish to update the events to include more data or otherwise change things. It is likely however that you do not wish to break things and you will not be able to update all event generators and consumers atomically. This is where versioning is important.

The key to making this work is compatibility, and that is not hard to achieve if a few simple rules are followed.

1. Events can never remove or change the meaning of existing fields. If you need to so this, you need to make new events that are logically independent from the old events.
2. If you wish to add new data to an existing event (The most common versioning operation), make sure you add it to the END of the existing arguments. This way the serialized format for the old fields is IDENTICAL and can be parsed by logic that has no understanding of the new data.
3. When you add data in this way, you should increment the version number for the event. You can do this for an EventSource by adding an attribute like [Event(ID, Version=1)]. When you don’t provide a version number EventSource defaults it to 0, so you can start with Version=1 for the first update. You can have at most 255 versions (so don’t version unless you need to).

If you follow these rules, TraceEvent can sort everything out for both old and new events. Old processing code will continue to see all the old properties even when the data was generated with an updated EventSource (because of rules (1) and (2)), and new code accessing old fields can use the payload length to notice that old EventSources have not emitted a field and thus can return a default value (0 or “”). Higher level logic that uses TraceEvent can always probe the Version fields associate with an event and do special handling but often the defaults provided by TraceEvent’s parsers are sufficient.

# Higher level Processing: TraceLog

So far the only way to process data in the TraceEvent library is using the ‘push’ model where you subscribe to events and get a callback when they arrive (either through C# events or IObservables). There are two reasons that the push model is the ‘fundamental’ mechanism in TraceEvent

1. The push model works for real time scenarios. The pull model (e.g. foreach) simply does not work in the real time case since real time lists have no end.
2. The push model naturally handles a heterogeneous list of events. Each callback gets a strongly typed data specific for that event. In a pull model (e.g. foreach), as each event is processed it must be cast to the correct type to get at its event-specific fields. This is clumsy and inefficient.

Thus the pull (callback) model is encouraged, but there are definitely cases where a pull model is convenient (e.g. iterating over all events of a particular type in an ETL file) where the pull model (foreach) is not problematic and the library should support.

Perhaps more importantly, the ‘TraceEventSource’ model’s simplicity can also be a problem. It is not uncommon to have millions of events and the callback model really only understand one way of accessing the data – enumerating it from start to finish (well you can abort part way). For many situation you would like to organize the data in a way that is closer to how the data will be used. For example it would be useful to able to find all the process, or threads, or modules or other things without having to read every last event in the file. It would also be useful to have these things be in ‘tables’ of their own (that are homogeneous), and many ways of accessing them (by name, or by time…).

Solving these problems is what the Microsoft.Diagnostics.Tracing.TraceLog class is all about. Unlike ETWTraceEventSource (which only supports the ‘push ’ (callback) model) and supports real time streams. TraceLog ONLY deals with files, and can provide both a push (callback) model as well as a ‘pull’ (foreach) model. It also contains a rich object model that allows you to

1. Efficiently access a range of events in time without having to read through from the beginning (random access).
2. Query the set of processes, threads, and modules mentioned in the trace.
3. It gives every item an index (small integer with a known bound) that uniquely identifies this. This allows you to create ‘side’ arrays that allow you to ‘hand’ additional information on the item.
4. Processes the raw stack trace information (which is a list of physical addresses) with the module load events and JIT compile events to form symbolic stack traces.

In short, the TraceLog class preforms the next level of common semantic interpretation of the event data, as well preserving the ability to access the raw event data.

##  Creating a TraceLog (an ETLX file).

TraceLog needs to remember not only the even information but also the processed form for the process, threads, modules and call stacks in the trace. It would have been since to ‘extend’ the ETL file format but that format is private and tuned for event logging, not lookup. Thus a new format was needed, which is called ETLX. Thus the basic architecture of processing is to take a ETL file and ‘convert’ it to a ETLX file. The TraceLog class is then simply the programmatic interface to this ETLX data. Because users typically begin with a ETL file, the TraceLog class has a ‘OpenOrConvert’ method that makes starting with an ETL file easy. Here is a simple program that prints out every event in an ETL file as XML.

using Microsoft.Diagnostics.Tracing;

using Microsoft.Diagnostics.Tracing.Etlx;

using System;

class Program {

 static void Main() {

 using (var traceLog = TraceLog.OpenOrConvert("MyFile.etl"))

 {

 foreach (TraceEvent data in traceLog.Events) {

 Console.WriteLine("Got Event {0}", data);

 }

 }

 }

}

The way this works is that OpenAnConvert assumes that every ETL file has an ETLX file associated with it (by changing the extension). You can give either and it will check if the ETLX file is up to date (newer than the corresponding ETL file). If it is it uses it, otherwise it generates it from the ETL file. In either case it returns the TraceLog that represents the up-to-date ETLX file data.

Once we have a TraceLog we can get at the original events using the ‘Events’ property that returns an IEnumerable<TraceEvent>. Thus you can do a ‘foreach’ on this and in our case, print (ToString) each event to get the XML.

But the power of the TraceLog file is that is has also computed summaries of useful information like the processes, and has integrate this all into a convenient object model. Here is code that prints the XML events but only for process of a given name and rolls it up by Thread.

using Microsoft.Diagnostics.Tracing;

using Microsoft.Diagnostics.Tracing.Etlx;

using System;

class Program {

 static void Main() {

 using (var traceLog = TraceLog.OpenOrConvert("MyFile.etl"))

 {

 var process = traceLog.Processes.FirstProcessWithName("PerfView");

 Console.WriteLine("Process devenv Start {0:f3} end: {0:f3}",

 process.StartTimeRelativeMsec, process.EndTimeRelativeMsec);

 foreach (var thread in process.Threads)

 {

 Console.WriteLine("Thread {0}", thread.ThreadID);

 foreach (var data in thread.EventsInThread)

 Console.WriteLine(" Event {0}", data);

 }

 }

 }

}

Notice that it was trivial to find all the data associated with a particular process (a common operation) and filter to just that. Further find the thread, and look at events only on those particular threads. Notices that It could print the start and end time (information that can only be formed by combining data from two events) trivially. Also you effectively are passing over the data multiple times (for each thread) and each time only getting the filtered data. Tracelog give you random access to the data.

One useful way to exploit the random access capability is to enumerate events backwards in time. This may seem like a strange thing to do, but it is quite useful because it is often the case that after you find a ‘bad event’ you need to search backwards for its cause. Doing this we the callback model is simply problematic, but with TraceLog’s object model it is pretty straightforward. In the example below we have reason to believe that an exception might be caused by the JIT compilation that occurred earlier in the trace. This code could be written to find interesting cases of this bug.

using Microsoft.Diagnostics.Tracing;

using Microsoft.Diagnostics.Tracing.Etlx;

using Microsoft.Diagnostics.Tracing.Parsers.Clr;

using System;

class Program {

 static void Main() {

 using (var traceLog = TraceLog.OpenOrConvert("MyFile.etl")) {

 // For a particular process

 var p = traceLog.Processes.FirstProcessWithName("perfView");

 // Find all the exceptions.

 foreach (var e in p.EventsInProcess.ByEventType<ExceptionTraceData>()) {

 Console.WriteLine("Exception {0}", e.ExceptionMessage);

 // See what method was JIT compiled just before it.

 var prevInproc = p.EventsInProcess.FilterByTime(p.StartTime, e.TimeStamp);

 foreach (var j in prevInproc.Backwards().ByEventType<MethodJittingStartedTraceData>())

 {

 Console.WriteLine("Last Jit {0}", j.MethodName);

 break;

 }

 }

 }

 }

}

In the code above we first find all the events in the ‘PerfVIew’ process and look for exception events (all pretty straightforward so far. When we find an exception we use the ‘FilterByTime’ operation to isolate just the events up to that point. We then apply (the rather magical), ‘Backwards’ operator that returns a collection that is the reversed time image of those events, and filter them to just the JIT events. We fine the first such event and print the information we need.

What is amazing about this is that it is efficient. All the intermediate collections are never actually formed, so what actually happens at runtime is that you start with the exception event walking backwards until you find the JIT event. Typically you find the event you are looking for very quickly (it is nearby, and you can always place a limit on how far you look back), and everything finishes quickly. It is handy.

## Using Call Stacks with the TraceEvent library

Perhaps the most compelling reason to use the TraceLog class is that it support symbolic resolution of stack information associated with events. ETW has the ability to collect stack traces associated with most events, but what actually gets logged are arrays of method return addresses. In addition some stacks are decoupled from the event they are associated with and may even be in two pieces (a kernel piece and a user mode piece) In addition these addresses need to be resolved to the module or JIT compiled method they belong to and the physical address needs to be resolved to a symbolic name. All this complexity make it infeasible use the raw stack events ‘on the fly’ but the ETLX conversion does all the necessary computation to make getting these stacks relatively easy. At least currently this functionality is only available via the TraceLog class. When and ETL file is converted to ETLX the stack events are processed into a efficient and compact form and wired up to the appropriate events.

There is a complete example of using the TraceLog class to access stack traces in the [TraceEvent code samples](http://www.nuget.org/packages/Microsoft.Diagnostics.Tracing.TraceEvent.Samples) but the actual API is pretty straightforward. Here is some code that opens an ETL file finds all the Exception events in the ‘PerfView’ process in the trace, and prints each method name

using Microsoft.Diagnostics.Tracing;

using Microsoft.Diagnostics.Tracing.Etlx;

using Microsoft.Diagnostics.Tracing.Parsers.Clr;

using System;

class Program {

 static void Main() {

 using (var traceLog = TraceLog.OpenOrConvert("MyFile.etl")) {

 var p = traceLog.Processes.FirstProcessWithName("perfView");

 foreach (var ex in p.EventsInProcess.ByEventType<ExceptionTraceData>()) {

 Console.WriteLine("Exception {0}", ex.ExceptionMessage);

 var cs = ex.CallStack();

 if (cs != null) {

 Console.WriteLine(" Stack: ", cs);

 while (cs != null) {

 Console.WriteLine(" Method: {0}",

 cs.CodeAddress.FullMethodName);

 cs = cs.Caller;

 }

 }

 }

 }

 }

}

As you can see the basic logic is pretty simple. TraceEvents have a ‘CallStack()’ method that returns a callstack (null if there is no stack), and each callstack has among other things a ‘FullMethodName’ property and a ‘Caller’ property, and you can keep calling ‘Caller’ until you have enumerate the complete stack up to the top of the thread. The model is significantly richer than is outlined here (you can actually get at line number and source file information), but you get the idea, it is pretty useful.

## The Ugly Details of Stack Resolution

While the user model for dealing with stack capture is pretty straightforward, unfortunately the underlying mechanism that enables it is pretty complex and fragile so it is easy for it to not work or only work partially. This is why it is a VERY good idea to start with working example (like the example in the [TraceEvent code samples](http://www.nuget.org/packages/Microsoft.Diagnostics.Tracing.TraceEvent.Samples)) and make sure that you have done all the required steps. PerfView is also useful here, because it is also an application that is doing ‘all the right things’ so it is good to confirm that it is working for PerfView before you start trying to debug why your code is not working properly.

Below are the steps in converting logging an event with a stack to a resolved symbolic name annotated with some things that can go wrong along the way.

1. When the event is logged the ETW system tries to craw the stack at runtime. However this can fail for various reasons
	1. On 32 bit machine the crawler assumes the compiler stores unwinding information (EBP frames) on the stack. If the compile does not do this the stack ‘breaks’ (can’t be unwound) and you lose any frames ‘toward thread start’.
	2. On 64 bit processes on 64 bit machines the crawler needs ‘unwind’ information. For native code this is stored in the EXE, but for Just In Time (JIT) compiled code on Systems **before Windows 8 (or Win2012 server)** the ETW system did not know how to find this unwind information and break at the first frame with JIT compiled code. This is the most common reason for stack breakage, but will diminish machines are upgraded to new OSes.

If either of these happen there is nothing wrong with your code as it is an issue with the APP or with the ETW infrastructure. PerfView should have the same problem.

1. The raw addresses have to be associated with some entity that knows its symbolic name. At this point things work very differently for native code and code that is JIT compiled on the fly
	1. For native code, TraceEvent must fine DLL that includes the code, For this it needs information about all DLLs that where loaded in the process and what addresses they are loaded at. These are the Kernel ImageLoad events.
	2. For JIT compiled code it needs to know the code ranges of all JIT compiled methods. For this it needs special .NET or Jscript events specifically designed for this purpose.

If the necessary events are not present, the best that can be done is to show the address value as a Hexadecimal number (which is not very helpful). **Thus it is critical that these events be present.** Complicating this is the fact that in many scenario of long running processes. If the process lives longer than the collection interval, then there can be image loads or JIT compilation that occurred before the trace started. We need these events as well. To get them the ETW providers involved support something called ‘CAPTURE\_STATE’ which causes them to emit events for all past image loads or JIT compilations. **The logic for capturing data must explicitly include logic for triggering this CAPTURE\_STATE**.

1. For JIT compiled code, we are mostly done, however for native code, the symbolic name has only been resolved to the DLL level. To go further you need to get the mapping from DLL address to symbolic name. This is what the debugger PDB (program database) files do. For this you need to be able to find these PDB files. There are a number of things that can go wrong.
	1. You must set your \_NT\_SYMBOL\_PATH environment variable to locations where to search for the PDBS. If you do not you will only know the module and hex address.
	2. For operating system DLLs, the PDBS live on what is called a symbol server. To find these your \_NT\_SYMBOL\_PATH much include the name for these symbol servers (the public Microsoft symbol server is SRV\*http://msdl.microsoft.com/download/symbols). However to look up a DLL in the symbol server **you need a special GUID associated with the DLL, and a RAW ETL file does NOT INCLUDE this GUID**!. If you try to look up the DLL’s PDB on the machine where the DLL exists, TraceEvent can fetch the necessary GUID from the DLL itself, but if the ETL file was copied to another machine this will not work and the PDB cannot be fetched. Running the TraceEventSource.MergeInPlace operation rewrites the raw ETL file so that it includes the necessary DLL GUIDS and thus is a requirement if you move the data off the collection machine (and you want symbolic information for native code stacks.
	3. For .NET code all the library code is precompiled (NGENED) and so is looked up using a PDB like the native case. However unlikely native DLLs, the PDBS for the NGEN images typically do not saved on the Microsoft symbol server. Instead you must generate the PDBS for the NGEN images from the IL images as you need them. Again if you resolve the symbols on the machine where the collection happened, at the time you resolve the symbols TraceEvent’s SymbolReader class will automatically generate the NGEN image for you and cache it, however if you move the ETL file off the machine, you need to generate the NGEN PDBs as well as merge the ETL file to get the symbolic information for the .NET code in NGEN images. This is what the SymbolReader. GenerateNGenSymbolsForModule method can help you do. TODO MORE

So in summary to get good stacks and have them work on any machine for any code you need to

1. Collect the necessary ImageLoad and JIT compile events (including CAPTURE\_STATE so you get information about events that preceded data collection start)
2. Merge the raw ETL files so that PDB signature information is incorporated into ETL file.
3. Generate the necessary NGEN PDBs and upload them along with the ETL file to the machine where symbolic resolution will happen.

### Source Code Support

TODO

# Building Compile-time TraceEventParsers using TraceParserGen

If you have built your own EventSource so far the only way you have of accessing the data from your EventSource is to use the DynamicTraceEventParser class. As mentioned previously the experience codeing against DynamicTraceEventParser is not great since each property of the event has to be fetched with the ‘PayloadByName’ method and it is very easy to get the names wrong. What we want is a way of creating a specialized COMPILE TIME TraceEventParser that ‘knows’ about this ETW provider (EventSource). This is what the TraceParserGen.exe tool does. The process is really very simple

* All ETW providers, including EventSources, have a XML file called a ‘manifest’ that describes all the events and their properties.
* TraceParserGen takes a XML manifest and creates a C# file that defines a TraceEventParser for the provider described in the manifest.
* You can then include this C# file in your application and get a first class (and efficient) experience

In fact most of the parsers included in the TraceEvent library were generated using the TraceParserGen tool.

## Getting the XML Manifest

There are two main cases for ETW files that you may want TraceEventParsers for

1. An EventSource (which is not registered with the OS)
2. An ETW provider registered with the OS (something returned from logman query providers)

In both cases, the [PerfView](http://www.microsoft.com/en-us/download/details.aspx?id=28567) tool is an excellent way of getting the Manifest you need.

### Creating XML Manifests for EventSources

1. Collect an ETL file that events from the EventSource you want a manifest for. For example collect a trace for the Microsoft-Demos-MySource EventSource do

 perfView /onlyProviders=\*Microsoft-Demos-MySource EventSource collect

which produces the file ‘PerfviewData.etl.zip’

1. From this data file run the following command

 perfView /noGui userCommand DumpEventSourceManifests PerfVIewData.etl.zip

This command reads this data file and finds the manifests that the EventSource has logged to the ETL file and extracts them into a directory called ‘Etwmanifests’. You should find a file called Microsoft-Demos-MySource.manifest.xml in that directory which is the manifest you are trying to generate.

### Creating XML Manifests OS registered ETW providers

The operating system provides a wealth of ETW providers. You can get an idea using the command

 Logman query providers

Which gives a list. (See the section below on discovery for more). When you have found a provider you are interested in you can create its manifest by running the command

 perfView /nogui userCommand DumpRegisteredManifest ***PROVIDER\_NAME***

For example running the command

 perfView /nogui userCommand DumpRegisteredManifest Microsoft-Windows-Kernel-File

will generate a manifest file Microsoft-Windows-Kernel-File.manifest.xml that describes the event for the OS ETW provider Microsoft-Windows-Kernel-File.

## Converting to XML Manifest to a TraceEventParser for a Provider

TODO details on getting TraceParserGen.exe

Once you have the XML manifest file, the rest is trivial simply run

 TraceParserGen ***ManifestFileName***

And it will generate a corresponding C# file, which you can include in your application.

TODO EXAMPLE

# Event Provider Discovery

One interesting aspect of the lifecycle of an event logging session is Event Provider discovery. Simply put, if you don’t already ‘know’ that a provider exists, how do you find out? This is what the ‘TraceEventProvders’ class is about.

The first problem that hits almost immediately is the fact that the fundamental ID associated with a provider is a GUID not a name. Clearly humans would prefer a name, but how to do you get from a name to the GUID (which is what the OS APIs want). There is another problem in that each provider has 64 bit bit-vector of ‘keywords’ which define groups of events that you can turn on and off independently. How do we discover what our ‘keyword’ possibilities are for any particular provider?

Traditionally, this was solved by manifest publication. The idea is that a provider would compile its manifest to a binary form (using a tool called MC.exe), attach it to a DLL as a resource, and then run a utility called ‘wevtutil’ that will publish the manifest to the operating system. Thus the OS has a list of every published event provider and this list includes its name and descriptions of all its keywords. You can use the command

 logman query providers

To see a list of all registered providers and once you have the name of a particular provider (say Microsoft-Windows-Kernel-Process), you can get a description of its keywords by using the command

logman query providers Microsoft-Windows-Kernel-Process

The functionality of these two commands is also available in the TraceEventProviders class (see GetPublishedProviders and GetProviderKeywords).

There are a couple of problems with ETW’s registration scheme. The first is that it requires a step at ‘install’ time, which is problematic for programs that wish to keep an ‘xcopy’ deployment characteristic. Second, at least currently, publishing a manifest with wevtutil requires administrative permissions, which is even more problematic for many scenarios.

It is possible publish the manifest for an EventSource using the wevtutil mechanism (this is what the EventRegister tool does), however this does not solve

 EventSources will not show up on the lists above. As mentioned previously, because EventSources use a standard way of generating its provider Guid from its name, you can get the Guid from the name, but you can’t get the names and descriptions of the keywords for a particular EventSource (although you can turn them all on blindly which actually works pretty well).

## Publication vs. Registration

Publication is all putting provide schema information (the manifest) somewhere where event consumers can find it and is not really and is a very static activity (effectively some system wide database is updated). However when an EventProvider actually starts running (e.g. an EventSource is created), the provider needs to register itself with the operating system (using its provider ID). At this time the provider also registers a callback so that if at a later time a session enables a provider this callback is invoked to update the provider.

The operating system keeps track of the providers that have been registered (even if they have not been enabled), as well as a list of all provider that have been enabled by some session (even if there are no processes that actually have register that providers. This list is useful because it is a much smaller list of ‘interesting’ providers (those that you could turn on right now). You can get this list with the TraceEventProviders.GetRegisteredOrEnabledProviders method.

Perhaps more useful that this is to ask for a particular process what providers are actually registered in that process. You can get this information with the command line:

 Logman query providers –pid PROCESS\_ID

or programmatically with the TraceEventProviders.GetRegisteredProvidersInProcess(int) method.

Unfortunately when a provider registers with the operating system (the Win32 EvenRegister API), it only registers its provider Guid, and not its name. Thus the command above and the API above cannot give the name of the provider, just its Guid. For providers registered with the OS, this is not a problem since the name can be looked up (TraceEventProviders.GetProviderName), however this will not work for EventSources since they do not register their manifest. Because the EventSource is active, it is possible to actually ‘ask it’ for its manifest by opening a session (it will dump its manifest as part of its activation). However this does require more work. TODO INCOMPLETE.

# On the fly filtering AND Logging using ETWReloggerTraceEventSource

TODO