|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Strongly Typed Events (EasyETW)  **C:\Users\branbray\Desktop\VS_ParticleWave_White.png** | | | | | |
|  | **Spec Status:** Not Started | | | **Last updated:** 7/10/2012 12:48 AM | |
|  | | | | | |
| **Product Unit** | |  | **FEATURE TEAM** | |  |
|  | | | | | |
| **Program Manager** | | AbhMon | **Tester** | | GKraut |
| **Developer** | | MatEll | **Writer** | |  |

Strongly Typed Events provides developers an easy and declarative way to add instrumentation and diagnostic events to their code. These events are manifested, meaning that there is metadata about the events (how many there are and the types of data in their payload) so they can be consumed by third party tools. We also provide an easy way for developers to write their events to Event Tracing for Windows.

**Microsoft Corporation Technical Documentation License Agreement (Standard)**  
**READ THIS!** THIS IS A LEGAL AGREEMENT BETWEEN MICROSOFT CORPORATION ("MICROSOFT") AND THE RECIPIENT OF THESE MATERIALS, WHETHER AN INDIVIDUAL OR AN ENTITY ("YOU"). IF YOU HAVE ACCESSED THIS AGREEMENT IN THE PROCESS OF DOWNLOADING MATERIALS ("MATERIALS") FROM A MICROSOFT WEB SITE, BY CLICKING "I ACCEPT", DOWNLOADING, USING OR PROVIDING FEEDBACK ON THE MATERIALS, YOU AGREE TO THESE TERMS. IF THIS AGREEMENT IS ATTACHED TO MATERIALS, BY ACCESSING, USING OR PROVIDING FEEDBACK ON THE ATTACHED MATERIALS, YOU AGREE TO THESE TERMS.  
  
1. For good and valuable consideration, the receipt and sufficiency of which are acknowledged, You and Microsoft agree as follows:  
  
(a) If You are an authorized representative of the corporation or other entity designated below ("**Company**"), and such Company has executed a Microsoft Corporation Non-Disclosure Agreement that is not limited to a specific subject matter or event ("**Microsoft NDA**"), You represent that You have authority to act on behalf of Company and agree that the Confidential Information, as defined in the Microsoft NDA, is subject to the terms and conditions of the Microsoft NDA and that Company will treat the Confidential Information accordingly;  
  
(b) If You are an individual, and have executed a Microsoft NDA, You agree that the Confidential Information, as defined in the Microsoft NDA, is subject to the terms and conditions of the Microsoft NDA and that You will treat the Confidential Information accordingly; or  
  
(c)If a Microsoft NDA has not been executed, You (if You are an individual), or Company (if You are an authorized representative of Company), as applicable, agrees: (a) to refrain from disclosing or distributing the Confidential Information to any third party for five (5) years from the date of disclosure of the Confidential Information by Microsoft to Company/You; (b) to refrain from reproducing or summarizing the Confidential Information; and (c) to take reasonable security precautions, at least as great as the precautions it takes to protect its own confidential information, but no less than reasonable care, to keep confidential the Confidential Information. You/Company, however, may disclose Confidential Information in accordance with a judicial or other governmental order, provided You/Company either (i) gives Microsoft reasonable notice prior to such disclosure and to allow Microsoft a reasonable opportunity to seek a protective order or equivalent, or (ii) obtains written assurance from the applicable judicial or governmental entity that it will afford the Confidential Information the highest level of protection afforded under applicable law or regulation. Confidential Information shall not include any information, however designated, that: (i) is or subsequently becomes publicly available without Your/Company’s breach of any obligation owed to Microsoft; (ii) became known to You/Company prior to Microsoft’s disclosure of such information to You/Company pursuant to the terms of this Agreement; (iii) became known to You/Company from a source other than Microsoft other than by the breach of an obligation of confidentiality owed to Microsoft; or (iv) is independently developed by You/Company. For purposes of this paragraph, "Confidential Information" means nonpublic information that Microsoft designates as being confidential or which, under the circumstances surrounding disclosure ought to be treated as confidential by Recipient. "Confidential Information" includes, without limitation, information in tangible or intangible form relating to and/or including released or unreleased Microsoft software or hardware products, the marketing or promotion of any Microsoft product, Microsoft's business policies or practices, and information received from others that Microsoft is obligated to treat as confidential.   
  
2. You may review these Materials only (a) as a reference to assist You in planning and designing Your product, service or technology ("Product") to interface with a Microsoft Product as described in these Materials; and (b) to provide feedback on these Materials to Microsoft. All other rights are retained by Microsoft; this agreement does not give You rights under any Microsoft patents. You may not (i) duplicate any part of these Materials, (ii) remove this agreement or any notices from these Materials, or (iii) give any part of these Materials, or assign or otherwise provide Your rights under this agreement, to anyone else.  
  
3. These Materials may contain preliminary information or inaccuracies, and may not correctly represent any associated Microsoft Product as commercially released. All Materials are provided entirely "AS IS." To the extent permitted by law, MICROSOFT MAKES NO WARRANTY OF ANY KIND, DISCLAIMS ALL EXPRESS, IMPLIED AND STATUTORY WARRANTIES, AND ASSUMES NO LIABILITY TO YOU FOR ANY DAMAGES OF ANY TYPE IN CONNECTION WITH THESE MATERIALS OR ANY INTELLECTUAL PROPERTY IN THEM.  
  
4. If You are an entity and (a) merge into another entity or (b) a controlling ownership interest in You changes, Your right to use these Materials automatically terminates and You must destroy them.  
  
5. You have no obligation to give Microsoft any suggestions, comments or other feedback ("Feedback") relating to these Materials. However, any Feedback you voluntarily provide may be used in Microsoft Products and related specifications or other documentation (collectively, "Microsoft Offerings") which in turn may be relied upon by other third parties to develop their own Products. Accordingly, if You do give Microsoft Feedback on any version of these Materials or the Microsoft Offerings to which they apply, You agree: (a) Microsoft may freely use, reproduce, license, distribute, and otherwise commercialize Your Feedback in any Microsoft Offering; (b) You also grant third parties, without charge, only those patent rights necessary to enable other Products to use or interface with any specific parts of a Microsoft Product that incorporate Your Feedback; and (c) You will not give Microsoft any Feedback (i) that You have reason to believe is subject to any patent, copyright or other intellectual property claim or right of any third party; or (ii) subject to license terms which seek to require any Microsoft Offering incorporating or derived from such Feedback, or other Microsoft intellectual property, to be licensed to or otherwise shared with any third party.  
  
6. Microsoft has no obligation to maintain confidentiality of any Microsoft Offering, but otherwise the confidentiality of Your Feedback, including Your identity as the source of such Feedback, is governed by Your NDA.  
  
7. This agreement is governed by the laws of the State of Washington. Any dispute involving it must be brought in the federal or state superior courts located in King County, Washington, and You waive any defenses allowing the dispute to be litigated elsewhere. If there is litigation, the losing party must pay the other party’s reasonable attorneys’ fees, costs and other expenses. If any part of this agreement is unenforceable, it will be considered modified to the extent necessary to make it enforceable, and the remainder shall continue in effect. This agreement is the entire agreement between You and Microsoft concerning these Materials; it may be changed only by a written document signed by both You and Microsoft.

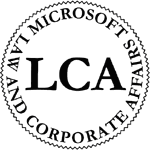


Table of Contents

1. Overview 1

1.1 Related Documents 1

1.2 Definitions and Reference 1

Customer Scenario and Context 1

2. Customer Experience 1

2.1 Basic Workflow 1

2.2 Instrumenting the Applciation 2

2.3 Customizing EventSources 3

2.4 End-to-End ETW Experience 4

2.5 Non-ETW Based Logging: EventListeners 4

3. Scope 5

3.1 Requirements 5

3.2 Non-requirements 6

4. Dependencies 6

5. Design 6

5.1 Basic Model 6

5.2 The Framework-provided ETW EventListeners. 7

5.2.1 EventSource Naming and GUIDs 8

5.3 The ETW Provider Manifest (EventSource Meta-data) 8

5.3.1 Non-Event methods in Event providers 9

5.3.2 Support for Enumerated types 9

5.3.3 User Defined Opcodes, Keywords, etc. 10

5.3.4 Pretty-printing events (message property) 10

5.3.5 Localization 11

5.3.6 .NET Localization Infrastructure 11

5.3.7 EventSource Localization 12

5.4 Self-Describing Event Logs 13

5.5 Extensibility Features (EventSource Commands) 13

5.6 Performance 14

5.6.1 Startup (EventSource creation) performance 14

5.6.2 EventSource Writing Performance 15

5.6.2.1 Improving performance when EventSource is Enabled 15

5.6.2.2 Creating fast WriteEvent overloads 16

5.7 Thead Safety 17

5.8 Surface Area 17

6. Feature Decisions / Q&A 19

7. Spec Completion Checklist 19

8. Change History 20

# Overview

This feature is for anyone who wishes to have their code to produce events about what the code is doing. Similar to the already existing [TraceSource](http://msdn.microsoft.com/en-us/library/system.diagnostics.tracesource.aspx) infrastructure, this feature allows developers to produce events. However unlike TraceSource, these events are strongly typed (given an event name you can describe its payload exactly) and statically discoverable (the list of events a program can produce can be determined via reflection).

In addition EventSources have built-in integration the windows [Event Tracing for Windows](http://msdn.microsoft.com/en-us/library/bb968803(VS.85).aspx) (ETW) infrastructure. ETW is a high performance eventing system for windows which is used extensively in both the windows kernel as well as many windows subsystems. These built-in ETW sources provide wonderful ‘low level’ value (indicating where CPU, Disk, and network activities occur), and EventSource builds on this by allowing programmers to mark ‘high level’ semantic actions (like a user command, or a high level event), and thus assign resource consumption to high level tasks.

## Related Documents

* [Event Tracing for Windows](http://msdn.microsoft.com/en-us/library/bb968803(VS.85).aspx) – Windows based infrastructure for logging events in a strongly typed way
* [ETW Manifest Schema](http://msdn.microsoft.com/en-us/library/aa384043(v=VS.85).aspx) – Description of the events particular ETW provider can generate.

## Definitions and Reference

* ETW – Event Tracing for Windows – a windows-based infrastructure for logging.
* ETW Provider – An entity in ETW that generates events
* ETW Controller – An entity in ETW that turns providers on and off.

# Customer Scenario and Context

Customers include

* The framework itself. We already have a private version of this code being used for the Task Parallel Library (System.Threading.Task.\*) as well as diagnostics for resource file lookup. We are also planning on using it for logging for the managed extensibility framework (MEF).
* Windows Communications Foundation (WCF). WCF previously used TraceSource but wishes to integrate with the Event Tracing for Windows (ETW) pipeline.
* Any users that currently use System.Diagnostics.TraceSource.

# Customer Experience

## Basic Workflow

Steve wishes to add ‘in the field’ logging to his application called **MyApp.exe**. The basic steps to do this are

1. Create a EventSource class called **MyAppEventSource** that has a method for each distinct type of event he wishes to log (details in next section)
2. Add calls to his code to these methods at the appropriate place.

At this point his code is instrumented and can be turned on in a variety of ways, including using the windows Event Tracing for Windows (ETW infrastructure). The basic workflow for using the event source using ETW is

1. Execute a tool that turns on the ETW provider (EventSource). All processes on the machine that have that provider (including processes that have already started) are activated by this command. You have control over where the data goes, and typically you ask the system to log it to a file (by convention a .ETL file).
2. Run the scenario.
3. Use the tool again to turn off the ETW provider (EventSource)
4. Process the ETL file to view the data or produce a report (often an XML file of some sort).

## Instrumenting the Applciation

Initially Steve only needs to log two events

1. A Load Event. This event also needs to log the name of the file loaded as well as the ‘ImageBase’ where the file was loaded, which acts as a ‘handle’ to uniquely identify the file in later events.
2. The Unload Event. This event needs to log the ‘ImageBase’ identifier which allows it to be matched up with the corresponding load event.

The code that Steve needs to write to generate these two events is only 4 lines of code:

sealed class MyAppEventSoruce: EventSource {

public static MyAppEventSource Log = new MyAppEventSource();

public void Load(long ImageBase, string Name) { if (IsEnabled()) WriteEvent(1, ImageBase, Name); }

public void Unload(long ImageBase) { if (IsEnabled()) WriteEvent(2, ImageBase); }

}

Essentially all that is required is to create a new class that inherits from ‘EventSource’. This class has set of methods, one per event to be created. Each method (event) can have data that is passed along with it (the method’s arguments). The class itself is a singleton and exposes its instance via the Log member;

// …

MyAppEventSoruce.Log.Load(0x700000, “MyFile”); // Log a load event

// …

MyAppEventSoruce.Log.Unload(0x700000); // Log a unload event

Notice the points at which events are fired (which are common operations), are very elegant. At each call site there are only the event source, the event name, and the actual data that had to be attached to the event being logged. In particular there is no information about the verbosity of the event, or bits that describe how the event is to be handled. We can do this because the events are STRONGLY TYPED. By using custom attributes we can ‘attach’ this information to the ‘Load’ method itself, and thus do not have to specify it at method Log time (improving both usability and efficiency).

Thus the purpose of subclasses like ‘MyAppEventSource’ is to provide a place where information specific to the user’s events (their names, arguments, verbosity etc), can be placed. Its job is also quickly forward events from this ‘strongly typed’ schema to helper methods (primary WriteEvent), provided by EventSource.

The code in these events methods is ‘boiler plate’ code, but thankfully it is very simple boiler plate. Each event method follows the same pattern.

Check to see if the provider is enabled. If so call WriteEvent, passing a provider unique small integer ‘EventId’ which identifies the particular event in the EventSource as well as all the arguments associated with the event (Events IDs start with 1)

Thus with just a few lines of code, Steve can instrument his code. He can then use ETW based tools like ‘PerfMonitor’ (described later) to turn on this logging and get output like the following.

<Event MSec= "1232.6356" PID="5868" PName= "(5868)" TID="2092" EventName="Load" ProviderName="MyAppEventSource" ImageBase="0x700000" Name="MyFile"/>

<Event MSec= "1238.3529" PID="5868" PName= "(5868)" TID="2092" EventName="Unload" ProviderName=" MyAppEventSource " ImageBase="0x700000"/>

Each event is logged with the precise time (to 100ns) it happened, as well as the process and thread that logged it, as well as the provider, event name, and any payload data. All of this information is strongly typed, which means that it can be processed mechanically by a sophisticated viewer, and also contains meta-data like the names of the fields in the payload which is necessary for the viewer to do a good job presenting the data.

## Customizing EventSources

More demanding users of EventSource will need additional control over exactly how an EventSource does its logging. For example users will want to indicate how verbose a particular logging message is, as well as assign events to various groups so that the right set of events can be selected easily. This additional information is provided by the developer by using custom attributes on the Event methods as well as the provider itself.

ETW has already devised a schema for this additional information and EventSource follows that. Verbosity in ETW is defined by the ‘Level’ of an event, and groups of events are formed by assigning each event a 64 bit bitvector called ‘Keywords’ which indicate which groups an events belongs to.

For example Steve might which to customize his EventSource as follows.

[EventSource(Name="MyCompany.MyApp")]

internal sealed class MyAppEventSource : EventSource

{

[Event(1, Level = EventLevel.Informational, Keywords=Keywords.Loader | Keywords.Critical)]

public void Load(long ImageBase, string Name) { if (IsEnabled()) WriteEvent(1, ImageBase, Name); }

[Event(2, Level = EventLevel.Verbose, Keywords = Keywords.Loader)]

public void Unload(long ImageBase) { if (IsEnabled()) WriteEvent(2, ImageBase); }

public class Keywords

{

public const EventKeywords Loader = (EventKeywords)0x0001;

public const EventKeywords Critical = (EventKeywords)0x0002;

}

}

In the example above Steve has defined two groups (Keywords), called ‘Loader’ and ‘Critical’. He has made the ‘Load’ event a member of both the ‘Loader’ and ‘Critical’ group, but the ‘Unload’ event is a member of just the ‘Loader’ group. He also marked the ‘Load’ event as being of ‘Informational’ verbosity, and the unload event as being of ‘Verbose’ verbosity. He has also decided to explicitly provide the ETW provider name ‘MyCompany.MyApp’ rather than letting it default to the simple name of the class.

Once marked in this way users at runtime can quite a bit of control over precisely which events get logged by indicating the verbosity level and keyword set desired.

## End-to-End ETW Experience

One of the expected ways to control EventSources at runtime is to use a ETW tool currently on CodePlex called [PerfMonitor](http://bcl.codeplex.com/wikipage?title=PerfMonitor&referringTitle=Home). PerfMonitor is really just a generic ETW Controller, but since EventSources are just ETW providers, it can be used to control EventSources.

For example, the command

* PerfMonitor /provider:#MyCompany.MyApp monitor **MyApp.exe <args>**

Invokes the ‘monitor’ command, which does the following:

1. Inform the operating system that the ‘MyCompany.MyApp’ provider should be turned on, and indicate that the output should go to a default (binary) file called ‘PerfMonitorOutput.etl’.
2. Run the ‘MyApp.exe’ with the given arguments
3. When the application completes, turn off logging (which closes the PerfMonitorOutput file)
4. Convert (print) the log file to XML as the file ‘PerfMonitorOutput.print.xml’

The result is an XML file that has elements like the following in it.

<Event MSec= "1232.6356" PID="5868" PName= "(5868)" TID="2092" EventName="Load" ProviderName=" MyAppEventSource" ImageBase="0x700000" Name="MyFile"/>

<Event MSec= "1238.3529" PID="5868" PName= "(5868)" TID="2092" EventName="Unload" ProviderName=" MyAppEventSource" ImageBase="0x700000"/>

The perfMonitor command line can be simplified even further, because by default ‘perfMonitor monitor’ will turn on every EventSource that can be found in the application by enumerating its static dependencies. Thus the command

* PerfMonitor monitor **MyApp.exe <args>**

is enough to gather data in many scenarios.

The key take-away, is that not only are EventSources easy to write (taking only a few lines of code to get started, but with the proper tool, it is also very easy to turn the logging on and off (making it actually easier than printf-style logging, however having all the advantages of strongly typed events and integration with the ETW infrastructure.

## Non-ETW Based Logging: EventListeners

By design EventSource follows the ETW model for its data ‘schema’. However there are times when ETW based logging is inappropriate. Today ETW is really designed to be used by administrators and collect data machine-wide. In a Silverlight scenario where the code is being run in the web browsers, administrative access is not possible, and the data should not be logged locally to the machine but rather sent back over the network to the host providing the code. Thus it is useful for EventSource to generalize the ‘back end’ of the logging mechanism to allow users control over where the events go.

This is supported by the ‘EventListener’ class. Conceptually an EventListener is an entity that receives logging events . An EventListener has ‘Enable’ and ‘Disable’ methods that allow it to subscribe to a particular EventSource and once subscribed, any events generated by that EventSource are routed to that EventListeners ‘OnEventWritten’ callback. At this point the Listener can do what he wishes with the event data.

For example, the IsolatedStorageEventListener shown below shows how to divert events to a simple text based file in isolated storage.

sealed class IsolatedStorageEventListener : EventListener {

private IsolatedStorageFile store;

private StreamWriter stream;

public IsolatedStorageEventListener(string location) {

store = IsolatedStorageFile.GetUserStoreForApplication();

stream = new StreamWriter(store.OpenFile(location, FileMode.OpenOrCreate));

}

protected override void OnEventWritten(EventWrittenEventArgs eventData) {

stream.WriteLine("Event {0} ", eventData.EventId);

foreach(object o in eventData.Payload) {

stream.WriteLine("\t{0}", o);

}

}

public override void Dispose() {

base.Dispose();

store.Dispose();

stream.Dispose();

}

}

A typical use might be

s\_listener = new IsolatedStorageEventListener(“myFile.dat”); // assign to a static variable

s\_listener.EnableEvents(MyAppEventSource.Log, EventLevel.Warning, MyAppEventSource.Keywords.Loader);

which creates a new IsolatedStorageEventListener that will write its events to the file ‘myFile.dat’ in isolated storage. It then subscribes to the MyAppEventSource.Log provider, but only events that have Verbosity ‘Warning’ or better and have the Loader keyword (group). For that point on, events from MyAppEventSource.Log will be directed to ‘myFile.dat’.

# Scope

ETW has limits on both the size and of an event and the data that can be encoded in an event. Unlike TraceSource which can trace any .NET object, ETW is limited to basic data types like integers of different sizes, strings and arrays of these types.

## Requirements

* Events write to ETW correctly as both manifest and classic providers.
* Feature is functional without including support for ETW so it can be included in CoreCLR.
* The system is usable in XCOPY deployment scenarios.
* The system has low overhead especially when it is turned off (pay for play).

## Non-requirements

* A tool that registers an event in the system so that consumers that do not expect the event manifest to be in the event stream itself can still consume the events. This may be an SDK tool or an Installer class that can be used with InstallUtil.
* A tool that can play back events to a set of TraceListeners in an out of process manner.
* A compatibility layer with TraceSource. This could be added out of band via the Listener model and would make a good CodePlex project.

# Dependencies

* This feature first appears in .NET Framework V4.5
* The ETW integration works best on Windows Vista and above, but still does work on XP.

*One pager section ends here*

# Design

## Basic Model

The basic model for event logging revolves around three classes

1. The EventSource Class: This base class is not intended to instantiated directly (in fact all its constructors are protected). It does however contain the bulk of the actual implementation and provides the helpers needed for subclasses to be as light as possible.
2. The user defined EventSource class. This is the class that holds the actually methods that log events, however each such method is specific to a particular event type. This class can be decorated with custom attributes that indicate any information that is common a particular event (like its verbosity, string to pretty-print it etc). Because users create this class, the goal is to make it as simple as possible.
3. The EventListener Class. The event Listener class is the class that receives events logged by all Events sources in the Appdomain. It is also responsible for dealing with the logic for turning EventSources on and off, and determining which events within an event source should be logged.

The model is as follows

* There are sets of EventSources and EventListeners in the appdomain.
* Every EventSource has filtering criteria FOR EACH LISTENER in the appdomain.
* Each listener gets to set filtering criteria, and that filtering criteria does not affect other listeners.
* Listeners only get the events they asked for.
* By default all event logging for Listeners are off. Listeners have to turn on events to receive them.

In a bit more detail

* Whenever EventSources get created, they add themselves to a (weak) appdomain-wide list of all active EventSources in the appdomain. When EventSources die (or are disposed), they removed themselves from the list.
* Whenver EventListeners get created they add themselves to a (strong) appdomain wide list of all active EventListeners. EventListeners must call Dispose() explicitly to remove themselves (since there is an implicit strong static reference).
* When EventSources get created they notify all EventListeners in the process that they now exist by calling the ‘OnEventSourceCreated’ callback. This gives every existing listener the opportunity to turn on logging for the new EventSource.
* When EventListerns get created, they receive a flurry of ‘catchup’ calls to ‘OnEventSourceCreated’ callbacks for each EventSource that currently exists in the appdomain.
* The invariant is that through the ‘OnEventSourceCreated’ callback, all events listeners know about every EventSource in the appdomain, and thus have an opportunity to turn on their events regardless of the ordering of creation of the EventSources and EventListeners.

The framework uses the ETW model for enabling and disabling events. Every event is given an ‘EventLevel’ and can be a member of up to 64 groups called ‘Keywords’. When you enable an event you give a level and a 64 bit integer representing a set of Keywords. If the event is no more verbose than the requested level (eg. It is a warning, and the requested level was Informational) ***and*** is a member of at least one of the specified groups, then the event is enabled for the given EventListener. For enabled events, the EventSource will call the ‘OnEventWritten’ method on the EventListener when a WriteEvent call is made.

Thus the general workflow is:

* An EventSource is created (off by default)
* An EventListener is created (this may be before or after the EventSource is created)
* The EventListener gets a ‘OnEventSourceCreated’ event
* The EventListener calls the EnableEvents method passing a particular EventSource. This enables events to be sent to the from that source to the listener.
* Code calls a logging method (on the subclass of EventSource)
* The event is passed to the OnEventWritten method on the EventListener
* The listener processes the event in an arbitrary fashion.
* Eventually (often at AppDomain shutdown as part of EventListener finalization) the EventListener.Dispose method is called and the EventListener disables logging from any event source it subscribed to.

## The Framework-provided ETW EventListeners.

Conceptually, the framework provides a built-in EventListener that sends its logging messages to the windows ETW infrastructure. We don’t really expect a lot of other listeners to be created. EventSoure is highly tuned for the ETW case. Ideally anyone who can use ETW should just use that.

The ETW listener is ‘built in’ because the simple act of creating an EventSource wirs into the ETW controller infrastructure. In particular, if OS ‘EnableTrace’ function is called with a GUID corresponding to the EventSource, then the EventSource will start calling the OS API to log events according to ETW rules. Thuys tools like ‘logman’ and ‘xperf’ can immediately be used to control the EventSource. Logically the framework did this by providing an ‘ETWEventListener’ but in actual implementation is more intertwined for efficiency.

### EventSource Naming and GUIDs

EventSource follows ETW conventions wherever possible. One such ETW convention is how ETW providers are named. In ETW the fundamental name for a provider is GUID. While conceptually GUIDs are nice, in actual practice GUIDs are a pain because no human being can be expected to remember a guild. Thus there needs to be a way to convert two and from this GUID to a more ‘friendly’ name. Normally ETW does this by requiring that provider ‘register’ themselves with the system, providing a manifest which includes a friendly name. However such registration requires ADMIN privileges (since it is a machine wide activity) and thus interferes with simplicity and XCOPY style deployment.

The solution EventSource chosen was to strongly encourage that an EventSource’s GUID be derived from its name. There already was a standard for forming a GUID from a string, [RFC4122](http://www.ietf.org/rfc/rfc4122.txt), so EventSource simply used that. The result is that in normal usage you don’t need to specify a GUID when defining an EventSource, nor when referring to an EventSource (for enlightened tools that know about the convention).

By default an EventSource’s Name is the name (without the namespace) of the class. This is OK for ‘private’ EventSources, but for more public ones it is better to have a hierarchical namespace (eg ComanyName.ProviderName). The EventSourceAttribute allows the ETW name to be defined independently of the name of the class.

The EventSourceAttribute also allows the user to explicitly specify a GUID for the EventSource. This allows Existing ETW providers to be upgraded to using EventSources, but generally its use is discouraged.

## The ETW Provider Manifest (EventSource Meta-data)

Fundamental to the ETW eventing model is the concept of event Meta-data call an [ETW Manifest](http://msdn.microsoft.com/en-us/library/aa384043(v=VS.85).aspx). Simply put, there is a bunch of information that is known about an event even before it is generated. This information includes

1. The name of the event
2. The provider (EventSource) that generates the event (providers are identified by a GUID).
3. A small number (event ID) that distinguishes that event from all other events generated from the same provider.
4. The names and types of all the fields in the payload
5. The verbosity (EventLevel) associated with the event
6. The version number of the event
7. The keywords (groups) that the event belongs to
8. A formatting string that can ‘pretty print’ the event data for human consumption.

There are also more ETW specific concepts such as

1. Events can be assigned an primary grouping called a ‘Task’. Tasks typically allow you logically group a set of events that come from a ‘subsystem’. Formally a task is a small integer that identifies this group. Unlike keywords, the upper bound on the number of tasks is quite large (64K).
2. Events can also be assigned a small integer value called an ‘Opcode’ that identifies the specific operation within a Task that is being logged. The value of Opcodes over event IDs is that there are some well defined ones like ‘Start’ and ‘Stop’ which allow tools to operate on events in a generic way (that is they know it is interesting to compute he time interval between such opcodes).

Some of this information may encoded in the ‘EventAttribute’ which can be placed on the methods representing the event. The framework picks defaults for all these attributes, so EventAttribute is optional, however conceptually every event has all of the attributes above.

A lot of information about the events can be determined simply by inspecting the signature of the event logging methods. The names of the arguments become the field names of the payload, and the types of arguments are attached to these fields. Currently we only support the following types

1. primitive types (Booleans, integers of various sizes, and floating point),
2. strings
3. Guid
4. Any Enumerated type (can be a [Flags] enumeration or normal).

In particular arrays or structures are not supported. Support for both structures and arrays is straightforward and could be added in the future if there is demand.

Thus by reflecting on the user-defined event source class, it is thus possible to generate a ‘full fidelity’ ETW manifest definition XML that describes the ETW provider that the EventSource represents. The ‘EventSource.GenerateManifest’ returns an XML string to this manifest. This allows EventSource’s to participate in existing ETW infrastructure that requires manifests.

### Non-Event methods in Event providers

EventSource assumes that every instance method that returns void is an event and thus attempts to generate meta-data for it. However it is often useful to have ‘helper’ methods on the event source that should not be included in the ETW manifest. This is what the NonEventAttribute is for. Here is a typical use.

[NonEvent]

public void DumpNode(Node node)

{

if (!IsEnabled(EventLevel.Verbose, EventKeywords.References))

return;

AssemblyReferences(node.ID, node.ExpensiveComputeReferences());

}

private void DumpNode(int nodeID, string references) { WriteEvent(5, nodeID, references); }

In this example the application has a concept called a ‘Node’ and we wish to log all the references to that node. However this is an expensive operation and we only wish to compute it if logging for References are enabled. Thus we create a ‘DumpNode’ method that takes a node (cheap), check to see if we need to do the expensive operation, and only then do the expensive operation. The NonEventAttribute is needed on the method that takes the ‘Node’ type to avoid an error trying to create a ETW manifest for that helper method.

### Support for Enumerated types

ETW manifests have a concept call a ‘map’ which is designed to support enumerated types. There are two kinds: a ‘valueMap’ for normal exclusive enumerated types and a ‘bitmap’ for flags that can be combined. This corresponds nicely to .NET enumerated types and enumerated types with the [Flags] attribute. When EventSource GenerateManifest is generating a manifest, if it encounters an event argument which is an enumerated type, it remembers to also emit the proper ‘bitmap’ or ‘valueMap’ into the manifest. In this way, ETW tools (like PerfMonitor) can decode enumerated types with full fidelity.

In event methods that have enumerated types as arguments, for efficiency sake it is good practice to cast the enumerated type to ‘int’ before calling WriteEvent. Here is an example

public void SendEnums(MyColor color, MyFlags flags) { WriteEvent(4, (int)color, (int)flags);

The reason for this is that the ‘WriteEvent’ has a ‘params object[]’ overload which is relatively inefficient (an arrays is allocated to hold the parameters and each argument is boxed and put into the array). In the example above, by casting the enumerations to int, it causes the type specific overload of WriteEvent to be used, which is much more efficient because it avoids the ‘params’ overload.

### User Defined Opcodes, Keywords, etc.

EventSource tries to stay as strongly typed, and ‘.NET like’ as possible. Thus keywords, opcodes, and tasks are given enumerated types (EventKeywords, EventOpcodes, EventTasks). When users wish to create new keywords opcodes, and tasks, they should do so by defining nested classes called ‘Keywords’ ‘Opcodes’ and ‘Tasks’ and define constant values for these. These values can then be used in the ‘Event’ Attribute to decorate particular events. An example of this is shown below

**sealed class AdvancedEventSource : EventSource**

**{**

**[Event(1, Keywords = Keywords.Loader | Keywords.Other, Task = Tasks.Run, Opcode = EventOpcode.Start)]**

**public void RunStart(long TaskId) { if (IsEnabled()) WriteEvent(3, TaskId); }**

**[Event(2, Task = Tasks.Run, Opcode = EventOpcode.Stop)]**

**public void RunStop(long TaskId) { if (IsEnabled()) WriteEvent(4, TaskId); }**

**[Event(3, Opcode=AdvancedEventSource.Opcode.MyOpcode)]**

**public void SetGuid(Guid MyGuid) { if (IsEnabled()) WriteEvent(5, MyGuid); }**

**// #Keywords (notice they are bitsets (flags))**

**public class Keywords**

**{**

**public const EventKeywords Loader = (EventKeywords)0x0001;**

**public const EventKeywords Critical = (EventKeywords)0x0002;**

**public const EventKeywords Other = (EventKeywords)0x0004;**

**}**

**// #Keywords (notice they are bitsets (flags))**

**public class Opcodes**

**{**

**public const EventOpcode MyOpcode = (EventOpcode)0x10;**

**}**

**// #Tasks**

**public class Tasks**

**{**

**public const EventTask Run = (EventTask)1;**

**}**

**}**

Note that it is important that the naming convention above is followed because when EventSource assumes this is the case when it is generating its manifest. For example, it assumes any keywords for the EventSource are constants declared in a nested class called Keywords. Thus if you wish for your keywords, opcodes, and tasks to show up in the manifest, then need to follow the convention above.

### Pretty-printing events (message property)

ETW supports the concept of an event format string called the event ‘Message’. The idea is that this is a printf style format string where substrings of the form %1, %2, … are replaced with the values in the payload. This is useful to provide a more user friendly output for the event. We support this on EventSource by adding the ‘Message’ property to EventAttribute, however this format string does not use %1, %2, but rather .NET conventions of {0}, {1}, …. The message property only affects the manifest that is EventSource.GenerateManifest generates. It is not otherwise used by EventSource.

For example the code

[Event(5, Message = "The SetGuid has a guid value of {0}.")]

public void SetGuid(Guid MyGuid) { WriteEvent(5, MyGuid); }

will cause perfMonitor to display:

<Event MSec= "2072.0133 … FormattedMessage="The SetGuid has a guid value of 0c836fd3-ee92-4301-bc13-d8943e0c1e77." … >

Thus EventSource can mimic ‘printf’ logging.

### Localization

Some of the ETW meta-data (event messages, tasks, keywords, maps) are designed to be human readable and as such should be localized. EventSource leverages standard .NET localization techniques to support ETW localization.

### .NET Localization Infrastructure

In .NET localization is supported through the [.NET Resource mechanism](http://msdn.microsoft.com/en-us/library/ms950960.aspx). This mechanism has two ‘tiers’. At the first tier, every .NET assembly can have a ‘resource’ section, which is logically a list of blobs with string names. These named blobs are called ManifestResources and can contain arbitrary binary data. One very common kind of resource blob is the result of compiling a ‘.Resx’ file using the ‘[resgen.exe](http://msdn.microsoft.com/en-us/library/ccec7sz1(VS.80).aspx)’ tool. The Resx block is again a container that contains named blobs and forms the second ‘tier’ of localization. Unlike ManifestResources, every entry in the Resx container is also has a .NET type associated with it.

The convention in .NET for localization is to give each string you wish to localized a name (key), and simply add the localized string to a Resx resource as the value for this key. At runtime the .NET ResourceManger class is used to look up the key and find the corresponding value. This allows an application to be tailored to a particular locale simply by modifying the embedded Resx resource rather than modifying the code.

Going one step further, the ResourceManager class has support for allowing the application to support multiple locales (languages) simultaneously by using [satellite assemblies](http://en.wikipedia.org/wiki/.NET_assembly#Satellite_assemblies). A .NET assembly name can also have a culture (language) associated with it. Thus a single ‘Foo.dll’ might have additional ‘satellite’ assemblies (en-uk\Foo.dll, en-us\Foo.dll …) The ResourceManager can use the thread’s current culture (Thread.CurrentThread.CurrentUICulture) to find the most appropriate satellite assembly to fetch the strings from.

The mechanics of using localized strings in Visual Studio are as follows

1. Add a new ‘Resource File’ to the project (this is a Resx file). When the project is built, the assembly associated with the project will now contain a ManifestResource called ‘***DefaultNameSpace.ResxFileName***.resources’ where ***DefaultNameSpace*** is the default namespace for the project (Project -> Properties -> Application -> DefaultNameSpace) and ***ResxFileName*** is the name of the Resx file (without the .resx extention)
2. Open the new resource file, and the ‘resource designer’ UI appears. This is essentially a datagrid that allows you to enter name (keys) and values (localized strings).

### EventSource Localization

EventSource uses the standard ResourceManager class to localize the strings it needs. To find a localized string the ResourceManager class needs three things

1. The assembly (DLL) on which to locate the resource
2. The ManifestResource name which essentially indicates which one of the potentially many ‘resx’ files embedded in the DLL. By convention in Visual Studio this is ‘***DefaultNameSpace.ResourceFileName***.resources’
3. A string key that identifies the particular string being localized.

EventSource finds these things by the following convention

1. The assembly (DLL) is the assembly in which the EventSource class is defined.
2. The ManifestResourceName is specified by the EventSourceAttribute.LocalizationResources placed on the EventSource. If this attribute property is not specified, then no string localization is done.
3. The localized strings EventSource needs are the localizations of events format strings (events), tasks, opcodes, keywords and enumeration constants (maps). Event source uses the following trivial conventions.

event\_***eventName***

task\_***taskName***

opcode\_***opcodeName***

keyword\_***keywordName***

map\_***mapName***

Where the names are the names used in the code for the event, task, opcode, keyword or map.

For example, let’s assume we have created a Visual Studio project with a default namespace called ‘MyApp’ and added to the project a resource file called ‘MyAppEventSoruceResources’. We could the localized the EventSource by simply adding a LocalizationResources property as follows:

[EventSoource(LocalizationResources=”MyApp.MyAppEventSourceResources.resources”)

sealed class MyAppEventSource: EventSource {

[Event(Keyword=”Loader”, Task=”Loading”)]

public void Load(long ImageBase, string Name) { WriteEvent(1, ImageBase, Name); }

public void Unload(long ImageBase) { WriteEvent(2, ImageBase); }

}

We could then add the following keys to the MyAppEventSoruceResources.Resx file:

event\_Load “File {1} is loading at address {0}”

event\_Unload “{0} is unloading.”

keyword\_Loader “Loader”

task\_Loading “Loading”

Now the ETW manifest that will be generated will include localized ‘message’ strings. By adding additional resource only satellite DLLs to the solution, the ETW logging can support multiple languages.

## Self-Describing Event Logs

While having meta-data associated with events is highly useful, it introduces the problem of how to find the meta-data when operating on the event logs. The mechanism standardized by the ETW infrastructure is to convert the meta-data to a binary form (using the MC.exe tool), attach it to a DLL as a resource, and then register that DLL with the operating system using a special (administrative), command (wevutil.exe). Unfortunately this has several important disadvantages

1. If the event data is moved to a different machine, the meta-data that is looked up may not be present or could be the wrong version. This could render the data useless (much like debugging without PDBs)
2. Registration would have to be done at deployment time, which means that XCOPY install is no longer possible. Worse, administrative privileges would be needed, increasing the administrative pain.

To avoid these problems EventSource outputs the meta-data associated with the EventSource as an event before the first event is fired. This way, the log contains the correct meta-data (guaranteed), and thus can be decoded on any machine. Moreover, it avoids the registration step, allow XCOPY deployment.

Because the manifest might exceed the ETW size limitation of 64K for a single events, the manifest is potentially dumped as ‘chunks’ that are under this size. When the provider is started it sends an event with

EventID = 0xFFFE, Task = 0, Opcode = FE

And a payload that consists of the following header followed by the next chunk of UTF8 data.

internal struct ManifestEnvelope

{

public const int MaxChunkSize = 0xFF00;

public enum ManifestFormats : byte

{

SimpleXmlFormat = 1, // simply dump the XML manifest as UTF8

}

public ManifestFormats Format;

public byte MajorVersion;

public byte MinorVersion;

public byte Magic; // 0x5B, just an unusual padding value.

public ushort TotalChunks;

public ushort ChunkNumber;

};

The result is that complete ETW log files always contain enough information to decode the data generated by EventSources. ETL files can be copied to other machines without problems. It also means that there is no strong reason to have to register an ETW provider in order to use it. XCOPY deployment is now possible.

## Extensibility Features (EventSource Commands)

The framework-provided filtering options (EventLevels and EventKeywords), are sufficient for many purposes, however it is also clear that there are scenarios where more is required. Typical examples are the desire to filter by Appdomain, or by a particular object type. There is also the desire to do other semantic actions (like dump the GC heap, or to do a GC, flush the working set, …) in response to an user command. To enable this EventSources support the ability to do arbitrary actions in response to commands. This is exposed through the EventSource.SendCommand method.

public static void SendCommand(EventSource eventSource, EventCommand command,

IDictionary<string, string> commandArguments)

The EventCommand enumerated type has a few pre-defined values (Which are all negative), the positive values are reserved for provider-specific commands. Along with the command comes a ‘key-value’ dictionary of arguments that are part of the command.

When a EventSource receives a command, it gets a callback on the OnEventCommand callback

protected virtual void OnEventCommand(EventCommandEventArgs command)

The arguments are passed through and thus the EventSource can do the action.

One common action would be to change the filtering. Filtering is tricky in that if you filter at the EventSource itself you will filter for ALL EventListeners, which is probably not what you want. Thus you need a way of selectively disabling logging for a particular EventListeners. This is what the EnableEvent and DisableEvent on the EventCommandEventArgs class are for.

public bool EnableEvent(int eventId)

public bool DisableEvent(int eventId)

These events take the ID of the event you with to enable or disable (the same number given as the first argument to the WriteEvent method). These methods will only enable or disable the EventListener that issued the command. These functions return a Boolean result that indicates whether ‘eventId’ is out of range. Thus you can turn off all events uniformly by calling DisableEvent starting with eventID 1 and continuing until DisableEvent returns false.

## Performance

EventSource was designed to be efficient; however there are still very useful guidelines to get the highest performance.

### Startup (EventSource creation) performance

EventSource does as little as possible at startup but it ALWAYS has to hook up to existing Listeners or ETW, which means

1. Calling the ETW EventRegister API to get a callback if an ETW command is made.
2. Place itself in the Appdomain wide list of EventSources that allow EventListeners to subscribe to it.

In the common case (no EventListeners), this overhead should be quite small, however it is still good practice to minimize the number of EventSources. Thus it is better to have one larger EventSource that it is to have two smaller ones. This leads to the following performance advice

* **Try to minimize the number of distinct EventSources. Logically related components that are deployed together should share an EventSource.**

### EventSource Writing Performance

Typically the main source of potential inefficiency is to during event writing. An important aspect of event writing performance is to be VERY small when the provider is off (pay for play). This leads to the following advice

* **Event methods should follow the pattern of doing a ‘IsEnabled() check as the first operation**

For example, you can see this pattern in the following ‘Load’ event method.

public void Load(long ImageBase, string Name) { if (IsEnabled()) WriteEvent(1, ImageBase, Name); }

The ‘IsEnabled()’ method is simply a field fetch on the EventSource class (1 instruction) and typically the event method itself (in this case Load()) is small enough to also be inlined by the JIT compiler, which means that when the calling load will only take 2 instructions (test and conditional jump) when the provider is off. However this 2 instruction minimum is on top of whatever it takes to fetch the EventSource. In .NET Static variables can be more expensive than instance variable fetches, so this leads to the following advice

* **(Pri 2) For high volume events, consider caching the EventSource as an instance method on an object that already needs to be fetched, rather than fetching the EventSource from a static.**

Thus instead of

MyEventSource.Log.Load(loader.fileLoaded); // log that a file was loaded.

Do

loader.logger = MyEventSource.Log; // Done once when loader is created

…

loader.logger.Load(loader.fileLoaded); // Done many times.

This optimization can save a small amount in some cases (NGEN or multi-appdomain cases) but is probably not worth it for low volume events.

#### Improving performance when EventSource is Enabled

A key aspect to the ‘WriteEvent’ method is that it is not one method but a collection of overloads.

void WriteEvent(int eventId, params object[] args);

void WriteEvent(int eventId, int arg1);

void WriteEvent(int eventId, long arg1);

void WriteEvent(int eventId, string arg1);

void WriteEvent(int eventId, int arg1, int arg2);

void WriteEvent(int eventId, long arg1, long arg2);

void WriteEvent(int eventId, string arg1, int arg2);

void WriteEvent(int eventId, string arg1, long arg2);

void WriteEvent(int eventId, string arg1, string arg2);

void WriteEvent(int eventId, int arg1, int arg2, int arg3);

void WriteEvent(int eventId, long arg1, long arg2, long arg3);

void WriteEvent(int eventId, string arg1, int arg2, int arg3);

void WriteEvent(int eventId, string arg1, string arg2, string arg3);

The basic idea is that if you match one of the type specific overloads then you use that (highly optimized) method, however if none of these match, the compiler will fall back to using the ‘params’ overload.

Unfortunately, the ‘params’ overload is relatively expensive, in particular it

1. Allocates an array to hold the variable arguments
2. Casts each parameter to an object (which causes allocations for primitive types)
3. Assigns these objects to the array
4. Calls the function, which then
5. Figures out the type of each argument, to determine how to serialize it to ETW

This is probably 10 to 20 times as expensive as specialized types. This probably does not matter much for low volume cases (because you only pay it when the provider is on, and low volume events simply don’t happen enough to matter), but for high volume events it can be important.

There are two important cases for insuring that the ‘params’ overload is not used

1. Insure that enumerated types are cast to ‘int’ so that they match one of the fast overloads.
2. Create new fast WriteEvent overloads for high volume payloads.

#### Creating fast WriteEvent overloads

EventSource exposes the protected method ‘WriteEventCore’ method to allow users to define new fast WriteEvent overloads. Making a new overload is not hard, but it does involve unsafe code. The basic procedure is to

1. stack allocate an array of ‘EventData’ descriptors that matches the number of payload items.
2. For each payload item set the correct size and value in the EventData structure
3. Call WriteEventCore with the initialized array.

Here is an example for adding a WriteEvent overload that takes four integer arguments.

[NonEvent]

public unsafe void WriteEvent(int eventId, int arg1, int arg2, int arg3, int arg4) {

EventData\* dataDesc = stackalloc EventProvider.EventData[4];

dataDesc[0].DataPointer = (IntPtr)(&arg1);

dataDesc[0].Size = 4;

dataDesc[1].DataPointer = (IntPtr)(&arg2);

dataDesc[1].Size = 4;

dataDesc[2].DataPointer = (IntPtr)(&arg3);

dataDesc[2].Size = 4;

dataDesc[3].DataPointer = (IntPtr)(&arg4);

dataDesc[3].Size = 4;

WriteEventCore(eventId, 4, (IntPtr)dataDesc);

}

## Thead Safety

The EventSource and EventListener are thread-safe in the sense that the framework itself does not require extra locking to work properly in a multi-threaded environment. Obviously, however, care must be taken in the callback routines if thread-shared state is accessed.

## Surface Area

namespace System.Diagnostics.Eventing {

public abstract class EventSource : IDisposable {

public string Name { get; }

public Guid Guid { get; }

public bool IsEnabled()

public bool IsEnabled(EventLevel level, EventKeywords keywords);

public void Dispose();

public void ToString();

// Protected Instance

protected EventSource();

protected virtual void OnEventCommand(EventListenerCommandEventArgs e);

protected void WriteEvent(int eventId)

protected void WriteEvent(int eventId, int arg)

protected void WriteEvent(int eventId, int arg1, int arg2)

protected void WriteEvent(int eventId, int arg1, int arg2, int arg3)

protected void WriteEvent(int eventId, long arg)

protected void WriteEvent(int eventId, long arg1, long arg2)

protected void WriteEvent(int eventId, long arg1, long arg2, long arg3)

protected void WriteEvent(int eventId, string arg)

protected void WriteEvent(int eventId, string arg1, string arg2)

protected void WriteEvent(int eventId, string arg1, string arg2, string arg3)

protected void WriteEvent(int eventId, string arg1, int arg2)

protected void WriteEvent(int eventId, string arg1, int arg2, int arg3)

protected void WriteEvent(int eventId, string arg1, long arg2)

protected void WriteEvent(int eventId, params object[] args)

protected void WriteEventCore(int eventId, int dataCount, EventSource.EventData\* data);

protected struct EventData {

public long DataPointer;

public int Size;

public int Reserved;

};

protected virtual void Dispose(bool disposing);

public static string GenerateManifest(Type eventSource,

string fullPathToBinaryWithEmbeddedManifest);

public static string GetName(Type eventSource);

public static Guid GetGuid(Type eventSource);

public static [IEnumerable](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://mscorlib:2.0.0.0:b77a5c561934e089/System.Collections.Generic.IEnumerable%3c%3e)<[EventSource](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://eventSupport:1.0.0.0/System.Diagnostics.Eventing.EventSource)> GetSources();

public static void SendCommand(EventSource eventSource, EventCommand command,

IDictionary<string, string> commandArguments)

}

[AttributeUsage(AttributeTargets.Method)]

public sealed class EventAttribute : Attribute {

public EventAttribute(int eventId);

public int EventId { get; }

public EventLevel Level { get; set; }

public EventKeywords Keywords { get; set; }

public EventOpcode Opcode { get; set; }

public EventTask Task { get; set; }

public byte Version { get; set; }

public string Message { get; set; }

}

[AttributeUsage(AttributeTargets.Method)]

public sealed class NonEventAttribute : Attribute {

public NonEventAttribute();

}

[AttributeUsage(AttributeTargets.Class)]

public sealed class EventSourceAttribute : Attribute

{

public EventSourceAttribute();

public string Name { get; set; }

public string Guid { get; set; }

public string LocalizationResources { get; set; }

}

public abstract class EventListener : IDisposable {

public EventListener();

public void EnableEvents(EventSource eventSource, EventLevel level);

public void EnableEvents(EventSource eventSource, EventLevel level,

EventKeywords matchAnyKeyword);

public void EnableEvents(EventSource eventSource, EventLevel level,

EventKeywords matchAnyKeyword, IDictionary<string, string> commandArguments);

public void DisableEvents(EventSource eventSource);

protected virtual void OnEventSourceCreated(EventSource eventSource);

protected abstract void OnEventWritten(EventWrittenEventArgs e);

// Returns a small dense integer ID for eventSoruce, Allows listeners to

// efficiently ‘attach’ data to an eventSource using List<T>.

static protected int EventSourceIndex(EventSource eventSource);

public void Dispose ();

protected virtual void Dispose(bool disposing);

}

public class EventListenerCommandEventArgs : EventArgs {

public EventListenerCommand Command { get; }

public IDictionary<String, String> Arguments { get; }

public void EnableEvent(Int32 eventId);

public void DisableEvent(Int32 eventId);

}

public class EventWrittenEventArgs : EventArgs {

public int EventId { get; }

public ReadOnlyCollection<object> Payload { get; }

public EventSource EventSource { get; }

public EventLevel Level { get; }

public EventKeywords Keywords { get; }

public EventTask Task { get; }

public EventOpcode Opcode { get; }

public Byte Version { get; }

}

public enum EventListenerCommand {

Update = 0,

SendManifest = -1,

Enable = -2,

Disable = -3

};

The following types already exist in the framework and are just here for reference.

public enum EventLevel {

LogAlways = 0,

Critical,

Error,

Warning,

Informational,

Verbose

}

public enum EventTask {

None = 0

}

public enum EventOpcode {

Info = 0,

Start,

Stop,

DataCollectionStart,

DataCollectionStop,

Extension,

Reply,

Resume,

Suspend,

Send,

Receive = 240

}

[Flags]

public enum EventKeywords : long {

None = 0x0,

WdiContext = 0x02000000000000,

WdiDiagnostic = 0x04000000000000,

Sqm = 0x08000000000000,

AuditFailure = 0x10000000000000,

AuditSuccess = 0x20000000000000,

CorrelationHint = 0x10000000000000,

EventLogClassic = 0x80000000000000

}

}

/