

# Event Tracing

Article • 01/07/2021 • 2 minutes to read

## Purpose

Event Tracing for Windows (ETW) provides application programmers the ability to start and stop event tracing sessions, instrument an application to provide trace events, and consume trace events. Trace events contain an event header and provider-defined data that describes the current state of an application or operation. You can use the events to debug an application and perform capacity and performance analysis.

This documentation is for user-mode applications that want to use ETW. For information about instrumenting device drivers that run in kernel mode, see [WPP Software Tracing](#) and [Adding Event Tracing to Kernel-Mode Drivers](#) in the Windows Driver Kit (WDK).

## Where applicable

Use ETW when you want to instrument your application, log user or kernel events to a log file, and consume events from a log file or in real time.

## Developer audience

ETW is designed for C and C++ developers who write user-mode applications.

## Run-time requirements

ETW is included in Microsoft Windows 2000 and later. For information about which operating systems are required to use a particular function, see the Requirements section of the documentation for the function.

## Process ETW traces in .NET code

You can use the [.NET TraceProcessing API](#) to analyze ETW traces for your applications and other software components. This API is used internally at Microsoft to analyze ETW data produced by the Windows engineering system, and it is also used to power several tables in [Windows Performance Analyzer](#). This API is available as a NuGet package.

For more information, see [this article](#).

# In this section

| Topic                                       | Description  |
|---|--|
| <a href="#">What's New in Event Tracing</a> | New features that were added to Event Tracing in each release.         |
| <a href="#">About Event Tracing</a>         | General information about Event Tracing.                               |
| <a href="#">Using Event Tracing</a>         | Task-related topics that describe how to use the ETW API.              |
| <a href="#">Event Tracing Reference</a>     | Detailed descriptions of ETW functions and other programming elements. |

# What's New in Event Tracing

Article • 01/07/2021 • 4 minutes to read

This section describes the new features that were added to Event Tracing for Windows in each release.

## Windows 10, version 1709

ETW can now optionally track binaries for all providers that are enabled to the session. The tracking applies retroactively for providers that were enabled to the session prior to the call, as well as for all future providers that are enabled to the session. You can also now query for the currently-configured maximum number of system loggers allowed by the operating system. For more information, see the [TraceProviderBinaryTracking](#) and [TraceMaxLoggersQuery](#) values of the [TRACE\\_INFO\\_CLASS](#) enumeration, as well as [Retrieving Additional Event Tracing Data](#).

ETW can now filter events based on event name. You can also determine which events get their stacks captured. For more information, see the [EVENT\\_FILTER\\_TYPE\\_EVENT\\_NAME](#), [EVENT\\_FILTER\\_TYPE\\_STACKWALK\\_NAME](#), and [EVENT\\_FILTER\\_TYPE\\_STACKWALK\\_LEVEL\\_KW](#) values of the [EVENT\\_FILTER\\_DESCRIPTOR](#) structure, as well as the associated [EVENT\\_FILTER\\_EVENT\\_NAME](#) and [EVENT\\_FILTER\\_LEVEL\\_KW](#) structures.

## Windows 10

[TraceLogging](#) builds on ETW and provides a simplified way to instrument code for native, .NET and WinRT developers. TraceLogging allows you to include structured data with events, correlate events, and does not require a separate instrumentation manifest XML file.

[Provider Traits](#) were added as a method of attaching more data to an individual provider registration. They can be used for manifest-based or TraceLogging providers. This currently includes support for adding a Provider Name and/or a Provider Group to an individual provider registration. Provider Groups are a new feature to allow multiple ETW providers to be controlled in aggregate by the group they belong to.

Periodic capture state is a way to allow capture state notifications to be routinely sent to providers. When this is enabled, notifications will only be sent to provider registrations that have been previously enabled to the current session. Each provider can define its

own response (if any) to a notification. For implementation details, see [TRACE\\_PERIODIC\\_CAPTURE\\_STATE\\_INFO](#).

## Windows 8.1 and Windows Server 2012 R2

The following features have been added to Event Tracing on Windows 8.1 and Windows Server 2012 R2.

Functions that support using event payload, scope, and stack walk filters used by the [EnableTraceEx2](#) function and the [ENABLE\\_TRACE\\_PARAMETERS](#) and [EVENT\\_FILTER\\_DESCRIPTOR](#) structures to filter on specific conditions in a logger session. For more information, see:

- [TdhAggregatePayloadFilters](#)
- [TdhCleanupPayloadEventFilterDescriptor](#)
- [TdhCreatePayloadFilter](#)
- [TdhDeletePayloadFilter](#)

In addition, see the extensively revised documentation for the [EnableTraceEx2](#) function and the [ENABLE\\_TRACE\\_PARAMETERS](#) and [EVENT\\_FILTER\\_DESCRIPTOR](#) structures that are used by these features.

A structure that defines an event payload filter predicate that describes how to filter on a single field in a trace session used by the new [TdhCreatePayloadFilter](#) function and a new structure used by event ID and stack walk filters. For more information, see:

- [EVENT\\_FILTER\\_EVENT\\_ID](#)
- [PAYLOAD\\_FILTER\\_PREDICATE](#)

Functions that retrieve information on events present in the provider manifest. For more information, see:

- [TdhEnumerateManifestProviderEvents](#)
- [TdhGetManifestEventInformation](#)

A structure that defines an array of events in a provider manifest used by the new [TdhEnumerateManifestProviderEvents](#) function. For more information, see:

- [PROVIDER\\_EVENT\\_INFO](#)

## Windows 8 and Windows Server 2012

The following features have been added to the Event Tracing on Windows 8 and Windows Server 2012.

Functions that performs operations on a registration object, provide event payload parsing, provide trace provider browsing, query event tracing session settings, and process a relogged trace file. For more information, see:

- [EventSetInformation](#)
- [TdhCloseDecodingHandle](#)
- [TdhGetDecodingParameter](#)
- [TdhGetWppProperty](#)
- [TdhGetWppMessage](#)
- [TdhLoadManifestFromBinary](#)
- [TdhOpenDecodingHandle](#)
- [TdhSetDecodingParameter](#)
- [TraceQueryInformation](#)

Interfaces that provide information to the relogger on the tracing process and when events are logged, access to data for a specific event, and access to relogger features that allow the manipulation of Event Trace Log (ETL) files. For more information, see:

- [ITraceEvent](#)
- [ITraceEventCallback](#)
- [ITraceRelogger](#)

Additional enumerations used by the new functions and interfaces. For more information, see:

- [EVENT\\_INFO\\_CLASS](#)
- [TRACE\\_QUERY\\_INFO\\_CLASS](#)

## Windows 7 and Windows Server 2008 R2

The following features were added in this release:

- The ability for providers to define filters in the manifest. In Windows Vista, controllers could pass filter data to the provider. However, the layout of the filter data was not defined in the manifest, so the provider would have to use other means to provide the filter definition to controllers. With this release, providers can define the filter definition in the manifest (see the **filters** attribute of the [ProviderType](#) complex type). Controllers can then use the [TdhEnumerateProviderFilters](#) function to determine the filter definition. Providers that use filters should use the [EventWriteEx](#) function to write the event.

- The ability to use a single buffer to gather events generated on multiple processors. Using a single buffer eliminates events from appearing out of order on multi-processors computers. For details, see the [EVENT\\_TRACE\\_NO\\_PER\\_PROCESSOR\\_BUFFERING](#) logging mode. By default, ETW uses per-processor buffers.
- The ability to capture a stack trace for events. To enable stack tracing for kernel events, see the [TraceSetInformation](#) function. To enable stack tracing for user events, see the [EVENT\\_ENABLE\\_PROPERTY\\_STACK\\_TRACE](#) flag for the [EnableProperty](#) member of [ENABLE\\_TRACE\\_PARAMETERS](#).
- The ability to specify the [EVENT\\_TRACE\\_BUFFERING\\_MODE](#) or [EVENT\\_TRACE\\_FILE\\_MODE\\_NEWFILE](#) logging mode with the [EVENT\\_TRACE\\_PRIVATE\\_LOGGER\\_MODE](#) logging mode (see [Logging Mode Constants](#)).
- The ability to enable a provider synchronously. By default, providers are enabled asynchronously. To enable a provider synchronously, set the *Timeout* parameter of [EnableTraceEx2](#).
- The ability for the controller to request that the provider log its state. For details, see the [EVENT\\_CONTROL\\_CODE\\_CAPTURE\\_STATE](#) flag for the *ControlCode* parameter of [EnableTraceEx2](#).
- The ability for consumers to format event data using the [TdhFormatProperty](#) function.
- The ability to decode manifested events on computers that do not contain the provider. For details, see the [TdhLoadManifest](#) function.

The following functions were added in this release:

- [EnableTraceEx2](#)
- [EventWriteEx](#)
- [TdhEnumerateProviderFilters](#)
- [TdhFormatProperty](#)
- [TdhLoadManifest](#)
- [TdhUnloadManifest](#)
- [TraceSetInformation](#)

The following structures were added in this release:

- [CLASSIC\\_EVENT\\_ID](#)
- [ENABLE\\_TRACE\\_PARAMETERS](#)
- [EVENT\\_EXTENDED\\_ITEM\\_STACK\\_TRACE32](#)
- [EVENT\\_EXTENDED\\_ITEM\\_STACK\\_TRACE64](#)
- [EVENT\\_FILTER\\_HEADER](#)
- [PROVIDER\\_FILTER\\_INFO](#)

The following enumerations were added in this release:

- [\*\*TRACE\\_INFO\\_CLASS\*\*](#)

The following MOF classes were added in this release:

- [\*\*StackWalk\*\*](#)
- [\*\*StackWalk\\_Event\*\*](#)

# About Event Tracing

Article • 01/07/2021 • 4 minutes to read

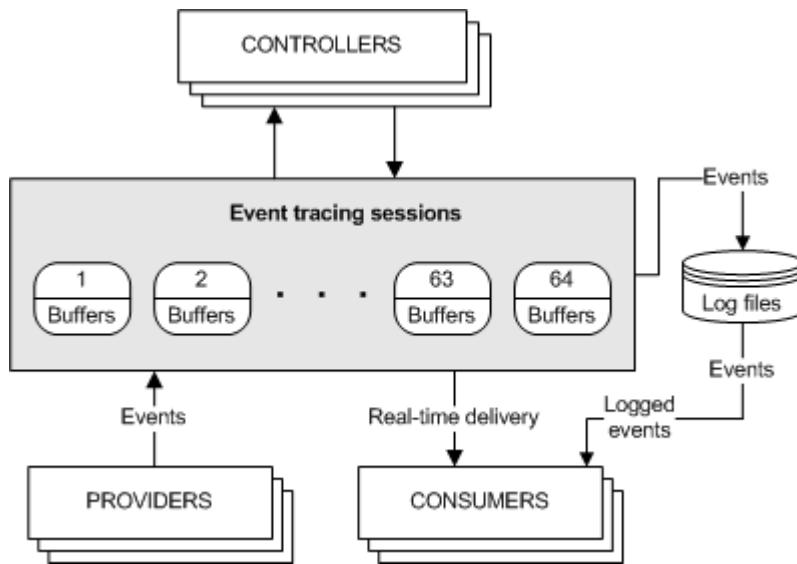
Event Tracing for Windows (ETW) is an efficient kernel-level tracing facility that lets you log kernel or application-defined events to a log file. You can consume the events in real time or from a log file and use them to debug an application or to determine where performance issues are occurring in the application.

ETW lets you enable or disable event tracing dynamically, allowing you to perform detailed tracing in a production environment without requiring computer or application restarts.

The Event Tracing API is broken into three distinct components:

- [Controllers](#), which start and stop an event tracing session and enable providers
- [Providers](#), which provide the events
- [Consumers](#), which consume the events

The following diagram shows the event tracing model.



## Controllers

Controllers are applications that define the size and location of the log file, start and stop [event tracing sessions](#), enable providers so they can log events to the session, manage the size of the buffer pool, and obtain execution statistics for sessions. Session statistics include the number of buffers used, the number of buffers delivered, and the number of events and buffers lost.

For more information, see [Controlling Event Tracing Sessions](#).

# Providers

Providers are applications that contain event tracing instrumentation. After a provider registers itself, a controller can then enable or disable event tracing in the provider. The provider defines its interpretation of being enabled or disabled. Generally, an enabled provider generates events, while a disabled provider does not. This lets you add event tracing to your application without requiring that it generate events all the time.

Although the ETW model separates the controller and provider into separate applications, an application can include both components.

For more information, see [Providing Events](#).

## Types of Providers

There are four main types of providers: MOF (classic) providers, WPP providers, manifest-based providers, and TraceLogging providers. You should use a manifest-based provider or a TraceLogging provider if you are writing applications for Windows Vista or later that do not need to support legacy systems.

### MOF (classic) providers:

- Use the [RegisterTraceGuids](#) and [TraceEvent](#) functions to register and write events.
- Use MOF classes to define events so that consumers know how to consume them.
- Can be enabled by only one trace session at a time.

### WPP providers:

- Use the [RegisterTraceGuids](#) and [TraceEvent](#) functions to register and write events.
- Have associated TMF files (compiled into a binary's .pdb) containing decoding information inferred from the preprocessor's scan of WPP instrumentation in source code.
- Can be enabled by only one trace session at a time.

### Manifest-based providers:

- Use [EventRegister](#) and [EventWrite](#) to register and write events.
- Use a manifest to define events so that consumers know how to consume them.
- Can be enabled by up to eight trace sessions simultaneously.

## TraceLogging providers:

- Use [TraceLoggingRegister](#) and [TraceLoggingWrite](#) to register and write events.
- Use self-describing events so that the events themselves contain all required information for consuming them.
- Can be enabled by up to eight trace sessions simultaneously.

All event providers fundamentally use the Event Tracing family of APIs ([TraceEvent](#) for legacy technologies and [EventWrite/EventWriteEx](#) for newer ones). Event providers simply differ in what field types they store in event payloads and where they store the associated event decoding information.

## Consumers

Consumers are applications that select one or more event tracing sessions as a source of events. A consumer can request events from multiple event tracing sessions simultaneously; the system delivers the events in chronological order. Consumers can receive events stored in log files, or from sessions that deliver events in real time. When processing events, a consumer can specify start and end times, and only events that occur in the specified time frame will be delivered.

For more information, see [Consuming Events](#).

## Missing Events

Perfmon, System Diagnostics, and other system tools may report on missing events in the Event Log and indicate that the settings for Event Tracing for Windows (ETW) may not be optimal. Events can be lost for a number of reasons:

- The total event size is greater than 64K. This includes the ETW header plus the data or payload. A user has no control over these missing events since the event size is configured by the application.
- The ETW buffer size is smaller than the total event size. A user has no control over these missing events since the event size is configured by the application logging the events.
- For real-time logging, the real-time consumer is not consuming events fast enough or is not present altogether and then the backing file is filling up. This can result if the Event Log service is stopped and started when events are being logged. A user has no control over these missing events.

- When logging to a file, the disk is too slow to keep up with the logging rate.

For any of these reasons, please report these problems to the provider of the application or service that is generating the events. These issues can only be fixed by the application developer or the service logging the events. If the missing events are being reported in the Event Log Service, this may indicate a problem with the configuration of the Event Log service. The user may have some limited ability to increase the maximum disk space to be used by the Event Log Service which may reduce the number of missing events.

# Event Tracing Sessions

Article • 01/07/2021 • 2 minutes to read

Event tracing sessions record events from one or more [providers](#) that a [controller](#) enables. The session is also responsible for managing and flushing the buffers. The controller defines the session, which typically includes specifying the session and log file name, type of log file to use, and the resolution of the time stamp used to record the events.

Event Tracing supports a maximum of 64 event tracing sessions executing simultaneously. Of these sessions, there are two special purpose sessions. The remaining sessions are available for general use. The two special purpose sessions are:

- Global Logger Session
- NT Kernel Logger Session

The Global Logger event tracing session records events that occur early in the operating system boot process, such as those generated by device drivers. For information on configuring the Global Logger event tracing session, see [Configuring and Starting the Global Logger Session](#).

The NT Kernel Logger event tracing session records predefined system events generated by the operating system, for example, disk IO or page fault events. For information on configuring the NT Kernel Logger event tracing session, [Configuring and Starting the NT Kernel Logger Session](#).

For information on defining a regular event tracing session, see [Configuring and Starting an Event Tracing Session](#).

**Windows 2000:** Supports only 32 event tracing sessions.

# Device Drivers

Article • 01/07/2021 • 2 minutes to read

This documentation is for user-mode applications that want to use ETW for event tracing. This documentation is not for device drivers that run in kernel-mode. For those writing device drivers and kernel-mode applications that want to use ETW as a low-impact debugger to troubleshoot timing-related problems, see [Event Tracing for Windows](#), [WPP Software Tracing](#), and [WMI Event Tracing](#) in the Windows Driver Kit (WDK) for information on instrumenting your driver.

# Using Event Tracing

Article • 01/07/2021 • 2 minutes to read

The following topics describe how to use the ETW API for event tracing.

| Topic   | Description   |
|---|---|
| <a href="#">Controlling Event Tracing Sessions</a>  | Describes how to manage event tracing sessions.   |
| <a href="#">Providing Events</a>                    | Describes how to register and instrument an event trace provider.   |
| <a href="#">Consuming Events</a>                    | Describes how to implement callback functions used to consume and process events from a trace log file or in real time. |
| <a href="#">Windows Software Trace Preprocessor</a> | Provides an efficient mechanism to log and consume events that occur during the execution of an application or driver.  |

Include the Wmistr.h header file before including the Evntrace.h header file.

# Controlling Event Tracing Sessions

Article • 01/07/2021 • 2 minutes to read

Event tracing sessions record events from one or more [providers](#). The controller defines the session and enables the providers. Defining the session typically includes specifying the session and log file name, type of log file to use, and the resolution of the time stamp used to record the events. Controllers can also update and query event tracing sessions.

The following topics demonstrate how to define and update a session, and enable event trace providers:

- [Configuring and Starting an Event Tracing Session](#)
- [Configuring and Starting a SystemTraceProvider Session](#)
- [Configuring and Starting an AutoLogger Session](#)
- [Configuring and Starting a Private Logger Session](#)
- [Updating an Event Tracing Session](#)
- [Retrieving Additional Event Tracing Data](#)

For information on flushing and querying sessions, see [ControlTrace](#) and [QueryAllTraces](#), respectively.

Only users running with elevated administrative privileges, users in the Performance Log Users group, and applications running as LocalSystem, LocalService or NetworkService can control event tracing sessions. To grant a restricted user the ability to control trace sessions, add them to the Performance Log Users group.

**Windows XP and Windows 2000:** Anyone can control a trace session.

# Configuring and Starting an Event Tracing Session

Article • 01/07/2021 • 4 minutes to read

To configure an event tracing session, use the [EVENT\\_TRACE\\_PROPERTIES](#) structure to specify the properties of the session. The memory you allocate for the **EVENT\_TRACE\_PROPERTIES** structure must be large enough to also contain the session and log file names that follow the structure in memory.

After you specify the properties of the session, call the [StartTrace](#) function to start the session. If the function succeeds, the *SessionHandle* parameter will contain the session handle, and the **LoggerNameOffset** property will contain the offset to the name of the session.

To enable the providers that you want to log events to your session, call the [EnableTrace](#) function to enable classic providers and the [EnableTraceEx](#) function to enable [manifest-based](#) providers. To enable providers that you want log events to your session filtering on specific conditions on Windows 8.1, Windows Server 2012 R2, and later, call the [EnableTraceEx2](#) function.

In addition, you can also trace additional information on an event with a call to the [TraceSetInformation](#) function. [TraceSetInformation](#) puts additional trace information into the extended data section of an event, and can include information such as the trace version info, or what providers are currently registered on the system. For more information, see [Retrieving Additional Event Tracing Data](#).

Up to eight trace sessions can enable and receive events from the same [manifest-based](#) provider. However, only one trace session can enable a [classic](#) provider. If more than one trace session tries to enable a classic provider, the first session would stop receiving events when the second session enables the provider. For example, if Session A enabled Provider 1 and then Session B enabled Provider 1, only Session B would receive events from Provider 1.

You can use any of the three functions to enable a provider but you may lose functionality if you use [EnableTrace](#) to enable a manifest-based provider because you will not be able to provide a MatchAllKeyword value, specify extended data items to include in the event, or provide provider-defined filter data. For more information, see the Remarks section of each function.

On Windows 8.1, Windows Server 2012 R2, and later, event payload, scope, and stack walk filters can be used by the [EnableTraceEx2](#) function and the

[ENABLE\\_TRACE\\_PARAMETERS](#) and [EVENT\\_FILTER\\_DESCRIPTOR](#) structures to filter on specific conditions in a logger session. For more information on event payload filters, see the [TdhCreatePayloadFilter](#), and [TdhAggregatePayloadFilters](#) functions and the [ENABLE\\_TRACE\\_PARAMETERS](#), [EVENT\\_FILTER\\_DESCRIPTOR](#), and [PAYLOAD\\_FILTER\\_PREDICATE](#) structures.

To determine the level and keywords used to enable a manifest-based provider, use one of the following commands:

- **Logman query provider-name**
- **Wevtutil gp provider-name**

The commands list the level and keywords only, the provider must document any filter data requirements for potential controllers.

To enumerate the manifest-based providers use **Wevtutil ep**.

For classic providers, it is up to the provider to document and make available to potential controllers the severity levels or enable flags that it supports. If the provider wants to be enabled by any controller, the provider should accept 0 for the severity level and enable flags and interpret 0 as a request to perform default logging (whatever that may be).

You can enable the provider before or after the provider registers itself. After enabling the provider, ETW will then call the provider's callback function. If the provider is not registered, ETW will call the provider's callback function after it registers itself.

You can also use the [EnableTrace](#) function to disable the provider (stop it from logging events to your session) or to update the logging level or enable flags of the provider. With the [EnableTraceEx](#) function, you can disable the provider or update the level, keywords, extended data and filter data. Each time you call the [EnableTrace](#) or [EnableTraceEx](#) function, ETW calls the provider's callback function. The provider remains enabled for the session until the session disables the provider.

To stop the trace session after collecting events, call the [ControlTrace](#) function and pass [EVENT\\_TRACE\\_CONTROL\\_STOP](#) as the control code. To specify the session to stop, you can pass the event tracing session handle obtained from an earlier call to the [StartTrace](#) function, or the name of a previously started session. Be sure to disable all providers before stopping the session. If you stop the session before first disabling the provider, ETW will disable the provider and try to call the provider's control callback function. If the application that started the session ends without disabling the provider or calling the [ControlTrace](#) function, the provider remains enabled.

If [ControlTrace](#) is successful, the session properties are updated to reflect the final property values and run statistics for the event tracing session.

For an example that starts an event tracing session, see the following:

- [Example that Creates a Session and Enables a Manifest-based Provider](#) - starts a trace session, enables a manifest-based provider, disables the provider and then stops the session.

For details on starting a trace session, see one of the following:

- [Configuring and Starting a SystemTraceProvider Session](#)
- [Configuring and Starting the NT Kernel Logger Session](#)
- [Configuring and Starting an AutoLogger Session](#)
- [Configuring and Starting the Global Logger Session](#)
- [Configuring and Starting a Private Logger Session](#)

## Related topics

[Configuring and Starting a Private Logger Session](#)

[Configuring and Starting a SystemTraceProvider Session](#)

[Configuring and Starting an AutoLogger Session](#)

[Configuring and Starting the NT Kernel Logger Session](#)

[ControlTrace](#)

[EnableTrace](#)

[EnableTraceEx](#)

[EnableTraceEx2](#)

[ENABLE\\_TRACE\\_PARAMETERS](#)

[EVENT\\_FILTER\\_DESCRIPTOR](#)

[EVENT\\_TRACE\\_PROPERTIES](#)

[PAYLOAD\\_FILTER\\_PREDICATE](#)

[StartTrace](#)

[TdhAggregatePayloadFilters](#)

## TdhCreatePayloadFilter

### Updating an Event Tracing Session

# Example that Creates a Session and Enables a Manifest-based or Classic Provider

Article • 01/07/2021 • 2 minutes to read

The following example shows how to start a trace session, enables a manifest-based or classic provider, disables the provider and then stops the session.

C++

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <strsafe.h>
#include <wmistr.h>
#include <evntrace.h>

#define LOGFILE_PATH L"<FULLPATHTOLOGFILE.etl>"
#define LOGSESSION_NAME L"My Event Trace Session"

// GUID that identifies your trace session.
// Remember to create your own session GUID.

// {AE44CB98-BD11-4069-8093-770EC9258A12}
static const GUID SessionGuid =
{ 0xae44cb98, 0xbd11, 0x4069, { 0x80, 0x93, 0x77, 0xe, 0xc9, 0x25, 0x8a,
0x12 } };

// GUID that identifies the provider that you want
// to enable to your session.

// {D8909C24-5BE9-4502-98CA-AB7BDC24899D}
static const GUID ProviderGuid =
{ 0xd8909c24, 0x5be9, 0x4502, { 0x98, 0xca, 0xab, 0x7b, 0xdc, 0x24, 0x89,
0x9d } };

void wmain(void)
{
    ULONG status = ERROR_SUCCESS;
    TRACEHANDLE SessionHandle = 0;
    EVENT_TRACE_PROPERTIES* pSessionProperties = NULL;
    ULONG BufferSize = 0;
    BOOL TraceOn = TRUE;

    // Allocate memory for the session properties. The memory must
    // be large enough to include the log file name and session name,
    // which get appended to the end of the session properties structure.

    BufferSize = sizeof(EVENT_TRACE_PROPERTIES) + sizeof(LOGFILE_PATH) +
```

```

    sizeof(LOGSESSION_NAME);
    pSessionProperties = (EVENT_TRACE_PROPERTIES*) malloc(BufferSize);
    if (NULL == pSessionProperties)
    {
        wprintf(L"Unable to allocate %d bytes for properties structure.\n",
        BufferSize);
        goto cleanup;
    }

    // Set the session properties. You only append the log file name
    // to the properties structure; the StartTrace function appends
    // the session name for you.

    ZeroMemory(pSessionProperties, BufferSize);
    pSessionProperties->Wnode.BufferSize = BufferSize;
    pSessionProperties->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
    pSessionProperties->Wnode.ClientContext = 1; //QPC clock resolution
    pSessionProperties->Wnode.Guid = SessionGuid;
    pSessionProperties->LogFileMode = EVENT_TRACE_FILE_MODE_SEQUENTIAL;
    pSessionProperties->MaximumFileSize = 1; // 1 MB
    pSessionProperties->LoggerNameOffset = sizeof(EVENT_TRACE_PROPERTIES);
    pSessionProperties->LogFileNameOffset = sizeof(EVENT_TRACE_PROPERTIES) +
    sizeof(LOGSESSION_NAME);
    StringCbCopy((LPWSTR)((char*)pSessionProperties + pSessionProperties-
    >LogFileNameOffset), sizeof(LOGFILE_PATH), LOGFILE_PATH);

    // Create the trace session.

    status = StartTrace((PTRACEHANDLE)&SessionHandle, LOGSESSION_NAME,
    pSessionProperties);
    if (ERROR_SUCCESS != status)
    {
        wprintf(L"StartTrace() failed with %lu\n", status);
        goto cleanup;
    }

    // Enable the providers that you want to log events to your session.

    status = EnableTraceEx2(
        SessionHandle,
        (LPCGUID)&ProviderGuid,
        EVENT_CONTROL_CODE_ENABLE_PROVIDER,
        TRACE_LEVEL_INFORMATION,
        0,
        0,
        0,
        NULL
    );

    if (ERROR_SUCCESS != status)
    {
        wprintf(L"EnableTrace() failed with %lu\n", status);
        TraceOn = FALSE;
        goto cleanup;
    }
}

```

```
wprintf(L"Run the provider application. Then hit any key to stop the
session.\n");
_getch();

cleanup:

    if (SessionHandle)
    {
        if (TraceOn)
        {
            status = EnableTraceEx2(
                SessionHandle,
                (LPCGUID)&ProviderGuid,
                EVENT_CONTROL_CODE_DISABLE_PROVIDER,
                TRACE_LEVEL_INFORMATION,
                0,
                0,
                0,
                NULL
            );
        }

        status = ControlTrace(SessionHandle, LOGSESSION_NAME,
pSessionProperties, EVENT_TRACE_CONTROL_STOP);

        if (ERROR_SUCCESS != status)
        {
            wprintf(L"ControlTrace(stop) failed with %lu\n", status);
        }
    }

    if (pSessionProperties)
    {
        free(pSessionProperties);
        pSessionProperties = NULL;
    }
}
```

# Configuring and Starting a SystemTraceProvider Session

Article • 06/02/2021 • 2 minutes to read

The SystemTraceProvider is a kernel provider with a predefined sets of kernel events supported on Windows 7, Windows Server 2008 R2, and later. On Windows 7 and Windows Server 2008 R2, the SystemTraceProvider could only be used for the NT Kernel Logger session.

On Windows 8, Windows Server 2012, and later, the SystemTraceProvider can be multiplexed for up to 8 logger sessions. The first two slots for logger sessions are reserved for the NT Kernel Logger and the Circular Kernel Context Logger .

For more information on using the NT Kernel Logger session as a trace provider, see [Configuring and Starting the NT Kernel Logger Session](#).

On Windows 10 SDK build 20348 and later, the SystemTraceProvider can be configured via separate System Providers, which can be controlled with [EnableTraceEx2](#) like standard Event Tracing for Windows event providers. For a full list of system providers, keywords, and corresponding legacy flags and groups, see [System Providers](#)

## Enable a SystemTraceProvider session

To enable the SystemTraceProvider to start a session other than the NT Kernel Logger, execute the following command:

```
tracelog -start MySession -f c:\Kernel1.etl -eflag PROC_THREAD+LOADER+CSWITCH
```

To programmatically enable the SystemTraceProvider to start a session other than the NT Kernel Logger, use the following steps.

- Define a private logger name.

```
#define PRIVATE_LOGGER_NAME L"Some Private Trace Session"
```

- At the controller, set the following members of the [EVENT\\_TRACE\\_PROPERTIES](#) structure.

Set **LogFileMode** to **EVENT\_TRACE\_SYSTEM\_LOGGER\_MODE**.

Set **LoggerName** to private logger, instead of **KERNEL\_LOGGER\_NAME**.

Make sure the `Wnode.Guid` member of the [EVENT\\_TRACE\\_PROPERTIES](#) structure is not set to `SystemTraceControlGuid`. You must assign a new **GUID** to this member.

- At the consumer, set the `LoggerName` member of the [EVENT\\_TRACE\\_LOGFILE](#) structure to this private logger.

 **Note**

If you want a non-administrators or a non-TCB process to be able to start a profiling trace session using the `SystemTraceProvider` on behalf of third party applications, then you need to grant the user profile privilege and then add this user to both the session **GUID** (created for the logger session) and the system trace provider **GUID** to enable the system trace provider. For more information, see the [EventAccessControl](#) function.

## Related topics

[Configuring and Starting a Private Logger Session](#)

[Configuring and Starting an AutoLogger Session](#)

[Configuring and Starting an Event Tracing Session](#)

[Configuring and Starting the NT Kernel Logger Session](#)

[System Providers](#)

[Updating an Event Tracing Session](#)

# Configuring and Starting the NT Kernel Logger Session

Article • 01/07/2021 • 2 minutes to read

The NT Kernel Logger session is an event tracing session that records a predefined set of kernel events. You do not call the [EnableTrace](#) function to enable the kernel providers. Instead, you use the [EnableFlags](#) member of [EVENT\\_TRACE\\_PROPERTIES](#) structure to specify the kernel events that you want to receive. The [StartTrace](#) function uses the enable flags that you specify to enable the kernel providers.

There is only one NT Kernel Logger session. If the session is already in use, the [StartTrace](#) function returns [ERROR\\_ALREADY\\_EXISTS](#).

For details on starting an event tracing session, see [Configuring and Starting an Event Tracing Session](#).

For details on starting a private logger session, see [Configuring and Starting a Private Logger Session](#).

For details on starting a Global Logger session, see [Configuring and Starting the Global Logger Session](#).

For details on starting an AutoLogger session, see [Configuring and Starting an AutoLogger Session](#).

The following example shows how to configure and start an NT Kernel Logger session that collects network TCP/IP kernel events and writes them to a 5MB circular file.

C++

```
#define INITGUID // Include this #define to use SystemTraceControlGuid in Evntrace.h.

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <strsafe.h>
#include <wmistr.h>
#include <evntrace.h>

#define LOGFILE_PATH L"<FULLPATHTOTHELOGFILE.etl>""

void wmain(void)
{
    ULONG status = ERROR_SUCCESS;
    TRACEHANDLE SessionHandle = 0;
```

```

EVENT_TRACE_PROPERTIES* pSessionProperties = NULL;
ULONG BufferSize = 0;

// Allocate memory for the session properties. The memory must
// be large enough to include the log file name and session name,
// which get appended to the end of the session properties structure.

BufferSize = sizeof(EVENT_TRACE_PROPERTIES) + sizeof(LOGFILE_PATH) +
sizeof(KERNEL_LOGGER_NAME);
pSessionProperties = (EVENT_TRACE_PROPERTIES*) malloc(BufferSize);
if (NULL == pSessionProperties)
{
    wprintf(L"Unable to allocate %d bytes for properties structure.\n",
BufferSize);
    goto cleanup;
}

// Set the session properties. You only append the log file name
// to the properties structure; the StartTrace function appends
// the session name for you.

ZeroMemory(pSessionProperties, BufferSize);
pSessionProperties->Wnode.BufferSize = BufferSize;
pSessionProperties->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
pSessionProperties->Wnode.ClientContext = 1; //QPC clock resolution
pSessionProperties->Wnode.Guid = SystemTraceControlGuid;
pSessionProperties->EnableFlags = EVENT_TRACE_FLAG_NETWORK_TCPIP;
pSessionProperties->LogFileMode = EVENT_TRACE_FILE_MODE_CIRCULAR;
pSessionProperties->MaximumFileSize = 5; // 5 MB
pSessionProperties->LoggerNameOffset = sizeof(EVENT_TRACE_PROPERTIES);
pSessionProperties->LogFileNameOffset = sizeof(EVENT_TRACE_PROPERTIES) +
sizeof(KERNEL_LOGGER_NAME);
StringCbCopy((LPWSTR)((char*)pSessionProperties + pSessionProperties-
>LogFileNameOffset), sizeof(LOGFILE_PATH), LOGFILE_PATH);

// Create the trace session.

status = StartTrace((PTRACEHANDLE)&SessionHandle, KERNEL_LOGGER_NAME,
pSessionProperties);

if (ERROR_SUCCESS != status)
{
    if (ERROR_ALREADY_EXISTS == status)
    {
        wprintf(L"The NT Kernel Logger session is already in use.\n");
    }
    else
    {
        wprintf(L"EnableTrace() failed with %lu\n", status);
    }

    goto cleanup;
}

wprintf(L"Press any key to end trace session ");

```

```
_getch();  
  
cleanup:  
  
    if (SessionHandle)  
    {  
        status = ControlTrace(SessionHandle, KERNEL_LOGGER_NAME,  
pSessionProperties, EVENT_TRACE_CONTROL_STOP);  
  
        if (ERROR_SUCCESS != status)  
        {  
            wprintf(L"ControlTrace(stop) failed with %lu\n", status);  
        }  
    }  
  
    if (pSessionProperties)  
        free(pSessionProperties);  
}
```

## Related topics

[Configuring and Starting a Private Logger Session](#)

[Configuring and Starting a SystemTraceProvider Session](#)

[Configuring and Starting an AutoLogger Session](#)

[Configuring and Starting an Event Tracing Session](#)

[Updating an Event Tracing Session](#)

# System Providers

Article • 09/21/2021 • 4 minutes to read

Beginning in Windows 10 SDK build 20348, the System Trace Provider's events can be enabled in the same way that other ETW providers are, with [EnableTraceEx2](#). The different flags and [group masks](#) associated with the System Trace Provider have been mapped to new trace providers, called System Providers, and matching keywords.

Like enabling the System Trace Provider directly, the System Providers can only be enabled by a session with the [EVENT\\_TRACE\\_SYSTEM\\_LOGGER\\_MODE](#) set.

## System Provider reference

- [System ALPC Provider](#)
- [System Config Provider](#)
- [System CPU Provider](#)
- [System Hypervisor Provider](#)
- [System Interrupt Provider](#)
- [System IO Provider](#)
- [System IO Filter Provider](#)
- [System Lock Provider](#)
- [System Memory Provider](#)
- [System Object Provider](#)
- [System Power Provider](#)
- [System Process Provider](#)
- [System Profile Provider](#)
- [System Registry Provider](#)
- [System Scheduler Provider](#)
- [System Syscall Provider](#)
- [System Timer Provider](#)

## System ALPC Provider

Provides events related to the ALPC system.

GUID: SystemAlpcProviderGuid {fcb9baaf-e529-4980-92e9-ced1a6aadfdf}

| Keyword                | Corresponding Legacy Flags and Groups |
|------------------------|---------------------------------------|
| SYSTEM_ALPC_KW_GENERAL | PERF_ALPC, EVENT_TRACE_FLAG_ALPC      |

## System Config Provider

Provides system configuration events.

GUID: SystemConfigProviderGuid {fef3a8b6-318d-4b67-a96a-3b0f6b8f18fe}

| Keyword                   | Corresponding Legacy Flags and Groups |
|---------------------------|---------------------------------------|
| SYSTEM_CONFIG_KW_SYSTEM   | PERF_SYSCFG_SYSTEM                    |
| SYSTEM_CONFIG_KW_GRAPHICS | PERF_SYSCFG_GRAPHICS                  |
| SYSTEM_CONFIG_KW_STORAGE  | PERF_SYSCFG_STORAGE                   |
| SYSTEM_CONFIG_KW_NETWORK  | PERF_SYSCFG_NETWORK                   |
| SYSTEM_CONFIG_KW_SERVICES | PERF_SYSCFG_SERVICES                  |
| SYSTEM_CONFIG_KW_PNP      | PERF_SYSCFG_PNP                       |
| SYSTEM_CONFIG_KW_OPTICAL  | PERF_SYSCFG_OPTICAL                   |

## System CPU Provider

Provides events related to the CPU.

GUID: SystemCpuProviderGuid {c6c5265f-eae8-4650-aae4-9d48603d8510}

| Keyword                    | Corresponding Legacy Flags and Groups |
|----------------------------|---------------------------------------|
| SYSTEM_CPU_KW_CONFIG       | PERF_CPU_CONFIG                       |
| SYSTEM_CPU_KW_CACHE_FLUSH  | PERF_CACHE_FLUSH                      |
| SYSTEM_CPU_KW_SPEC_CONTROL | PERF_SPEC_CONTROL                     |

## System Hypervisor Provider

Provides events relating to the hypervisor.

GUID: SystemHypervisorProviderGuid {bafa072a-918a-4bed-b622-bc152097098f}

| Keyword                       | Corresponding Legacy Flags and Groups |
|-------------------------------|---------------------------------------|
| SYSTEM_HYPERVISOR_KW_PROFILE  | PERF_HV_PROFILE                       |
| SYSTEM_HYPERVISOR_KW_CALLOUTS | PERF_HV_CALLOUTS                      |

| <b>Keyword</b>                  | <b>Corresponding Legacy Flags and Groups</b> |
|---------------------------------|--|
| SYSTEM_HYPERVISOR_KW_VTL_CHANGE | PERF_VTL_CHANGE                              |

## System Interrupt Provider

Provides events relating to DPCs and interrupts.

GUID: SystemInterruptProviderGuid {d4bbe17-b545-4888-858b-744169015b25}

| <b>Keyword</b>                      | <b>Corresponding Legacy Flags and Groups</b>  |
|-------------------------------------|---|
| SYSTEM_INTERRUPT_KW_GENERAL         | PERF_INTERRUPT,<br>EVENT_TRACE_FLAG_INTERRUPT |
| SYSTEM_INTERRUPT_KW_CLOCK_INTERRUPT | PERF_CLOCK_INTERRUPT                          |
| SYSTEM_INTERRUPT_KW_DPC             | PERF_DPC, EVENT_TRACE_FLAG_DPC                |
| SYSTEM_INTERRUPT_KW_DPC_QUEUE       | PERF_DPC_QUEUE                                |
| SYSTEM_INTERRUPT_KW_WDF_DPC         | PERF_WDF_DPC                                  |
| SYSTEM_INTERRUPT_KW_WDF_INTERRUPT   | PERF_WDF_INTERRUPT                            |
| SYSTEM_INTERRUPT_KW_IPI             | PERF_IPI                                      |

## System IO Provider

Provides events relating to multiple kinds of IO including disk, cache, and network.

GUID: SystemIoProviderGuid {3d5c43e3-0f1c-4202-b817-174c0070dc79}

| <b>Keyword</b>            | <b>Corresponding Legacy Flags and Groups</b>     |
|---------------------------|--|
| SYSTEM_IO_KW_DISK         | EVENT_TRACE_FLAG_DISK_IO                         |
| SYSTEM_IO_KW_DISK_INIT    | PERF_DISK_IO_INIT, EVENT_TRACE_FLAG_DISK_IO_INIT |
| SYSTEM_IO_KW_FILENAME     | PERF_FILENAME, EVENT_TRACE_FLAG_DISK_FILE_IO     |
| SYSTEM_IO_KW_SPLIT        | PERF_SPLIT_IO, EVENT_TRACE_FLAG_SPLIT_IO         |
| SYSTEM_IO_KW_FILE         | PERF_FILE_IO, EVENT_TRACE_FLAG_FILE_IO           |
| SYSTEM_IO_KW_OPTICAL      | PERF_OPTICAL_IO, EVENT_TRACE_FLAG_FILE_IO_INIT   |
| SYSTEM_IO_KW_OPTICAL_INIT | PERF_OPTICAL_IO_INIT                             |

| <b>Keyword</b>       | <b>Corresponding Legacy Flags and Groups</b> |
|----------------------|--|
| SYSTEM_IO_KW_DRIVERS | PERF_DRIVERS, EVENT_TRACE_FLAG_DRIVER        |
| SYSTEM_IO_KW_CC      | PERF_CC                                      |
| SYSTEM_IO_KW_NETWORK | PERF_NETWORK, EVENT_TRACE_FLAG_NETWORK_TCPIP |

Note: Enabling the SYSTEM\_IO\_KW\_DRIVERS keyword will automatically enable SYSTEM\_IO\_KW\_FILENAME as well. The SYSTEM\_IO\_KW\_FILENAME is also automatically turned on when the System Memory Provider is enabled with the SYSTEM\_MEMORY\_KW\_MEMORY keyword.

## System IO Filter Provider

Provides events related to IO filtering including timing and failures.

GUID: SystemIoFilterProviderGuid {fb09363-9e22-4661-b8bf-e7a34b535b8c}

| <b>Keyword</b>             | <b>Corresponding Legacy Flags and Groups</b> |
|----------------------------|--|
| SYSTEM_IOFILTER_KW_GENERAL | PERF_FLT_IO                                  |
| SYSTEM_IOFILTER_KW_INIT    | PERF_FLT_IO_INIT                             |
| SYSTEM_IOFILTER_KW_FASTIO  | PERF_FLT_FASTIO                              |
| SYSTEM_IOFILTER_KW_FAILURE | PERF_FLT_IO_FAILURE                          |

## System Lock Provider

Provides events relating to kernel locking mechanisms.

GUID: SystemLockProviderGuid {721ddfd3-dacc-4e1e-b26a-a2cb31d4705a}

| <b>Keyword</b>                   | <b>Corresponding Legacy Flags and Groups</b> |
|----------------------------------|--|
| SYSTEM_LOCK_KW_SPINLOCK          | PERF_SPINLOCK                                |
| SYSTEM_LOCK_KW_SPINLOCK_COUNTERS | PERF_SPINLOCK_CNTRS                          |
| SYSTEM_LOCK_KW_SYNC_OBJECTS      | PERF_SYNC_OBJECTS                            |

## System Memory Provider

Provides events relating to the memory manager.

| Keyword                        | Corresponding Legacy Flags and Groups                  |
|--------------------------------|--|
| SYSTEM_MEMORY_KW_GENERAL       | PERF_MEMORY  |
| SYSTEM_MEMORY_KW_HARDFAULTS    | PERF_HARDFAULTS,<br>EVENT_TRACE_FLAG_MEMORY_HARDFAULTS |
| SYSTEM_MEMORY_KW_ALLFAULTS     | PERF_ALLFAULTS,<br>EVENT_TRACE_FLAG_MEMORY_PAGEFAULTS  |
| SYSTEM_MEMORY_KW_POOL          | PERF_POOL  |
| SYSTEM_MEMORY_KW_MEMINFO       | PERF_MEMINFO   |
| SYSTEM_MEMORY_KW_PFBLOCK       | PERF_PFBLOCK   |
| SYSTEM_MEMORY_KW_MEMINFO_WS    | PERF_MEMINFO_WS  |
| SYSTEM_MEMORY_KW_HEAP          | PERF_HEAP  |
| SYSTEM_MEMORY_KW_WS            | PERF_WS  |
| SYSTEM_MEMORY_KW_CONTMEM_GEN   | PERF_CONTMEM_GEN                                       |
| SYSTEM_MEMORY_KW_VIRTUAL_ALLOC | PERF_VIRTUAL_ALLOC,<br>EVENT_TRACE_FLAG_VIRTUAL_ALLOC  |
| SYSTEM_MEMORY_KW_FOOTPRINT     | PERF_FOOTPRINT   |
| SYSTEM_MEMORY_KW_SESSION       | PERF_SESSION   |
| SYSTEM_MEMORY_KW_REFSET        | PERF_REFSET  |
| SYSTEM_MEMORY_KW_VAMAP         | PERF_VAMAP, EVENT_TRACE_FLAG_VAMAP                     |

#### Notes:

- Enabling the SYSTEM\_MEMORY\_KW\_MEMORY keyword will automatically enable SYSTEM\_IO\_KW\_FILENAME on the System IO Provider as well.
- The events enabled by the SYSTEM\_MEMORY\_KW\_POOL support a Pool Tag Filter, to selectively write events only for certain pool tags. This is configured as a [Schematized Filter](#) on the System Memory Provider. The PoolTag filter is constructed as follows:

```
C++
```

```
{  
EVENT_FILTER_HEADER Header;
```

```
ULONG PoolTags[ETW_MAX_POOLTAG_FILTER];  
}
```

- The [EVENT\\_FILTER\\_HEADER](#) should be initialized with Id set to SYSTEM\_MEMORY\_POOL\_FILTER\_ID and the size field set to the size of Header plus the used portion of the PoolTags array.
- Each Pool Tag is a 4 character string that is stored as a ULONG in the filter.

## System Object Provider

Provides events relating to the Object Manager.

GUID: SystemObjectProviderGuid {feb7460-3d1d-47eb-af49-c9eeb1e146f2}

| Keyword                 | Corresponding Legacy Flags and Groups |
|-------------------------|---------------------------------------|
| SYSTEM_OBJECT_KW_HANDLE | PERF_OB_HANDLE                        |
| SYSTEM_OBJECT_KW_OBJECT | PERF_OB_OBJECT                        |

## System Power Provider

Provides events relating to the system's power state.

GUID: SystemPowerProviderGuid {c134884a-32d5-4488-80e5-14ed7abb8269}

| Keyword                          | Corresponding Legacy Flags and Groups |
|----------------------------------|---------------------------------------|
| SYSTEM_POWER_KW_GENERAL          | PERF_POWER                            |
| SYSTEM_POWER_KW_HIBER_RUNDOWN    | PERF_HIBER_RUNDOWN                    |
| SYSTEM_POWER_KW_PROCESSOR_IDLE   | PERF_PROCESSOR_IDLE                   |
| SYSTEM_POWER_KW_IDLE_SELECTION   | PERF_IDLE_SELECTION                   |
| SYSTEM_POWER_KW_PPM_EXIT_LATENCY | PERF_PPM_EXIT_LATENCY                 |

## System Process Provider

Provides events relating to the process, including lifetime information, image load events, and thread related events.

GUID: SystemProcessProviderGuid {151f55dc-467d-471f-83b5-5f889d46ff66}

| <b>Keyword</b>                  | <b>Corresponding Legacy Flags and Groups</b>            |
|---------------------------------|---|
| SYSTEM_PROCESS_KW_GENERAL       | PERF_PROCESS, EVENT_TRACE_FLAG_PROCESS                  |
| SYSTEM_PROCESS_KW_INSWAP        | PERF_PROCESS_INSWAP                                     |
| SYSTEM_PROCESS_KW_FREEZE        | PERF_PROCESS_FREEZE                                     |
| SYSTEM_PROCESS_KW_PERF_COUNTER  | PERF_PERF_COUNTER,<br>EVENT_TRACE_FLAG_PROCESS_COUNTERS |
| SYSTEM_PROCESS_KW_WAKE_COUNTER  | PERF_WAKE_COUNTER                                       |
| SYSTEM_PROCESS_KW_WAKE_DROP     | PERF_WAKE_DROP  |
| SYSTEM_PROCESS_KW_WAKE_EVENT    | PERF_WAKE_EVENT   |
| SYSTEM_PROCESS_KW_DEBUG_EVENTS  | PERF_DEBUG_EVENTS,<br>EVENT_TRACE_FLAG_DEBUG_EVENTS     |
| SYSTEM_PROCESS_KW_DBGPRINT      | PERF_DBGPRINT, EVENT_TRACE_FLAG_DBGPRINT                |
| SYSTEM_PROCESS_KW_JOB           | PERF_JOB, EVENT_TRACE_FLAG_JOB                          |
| SYSTEM_PROCESS_KW_WORKER_THREAD | PERF_WORKER_THREAD                                      |
| SYSTEM_PROCESS_KW_THREAD        | PERF_THREAD, EVENT_TRACE_FLAG_THREAD                    |
| SYSTEM_PROCESS_KW_LOADER        | PERF_LOADER, EVENT_TRACE_FLAG_IMAGE_LOAD                |

## System Profile Provider

Provides profiling events.

GUID: SystemProfileProviderGuid {bfed0324-1cee-496f-a409-2ac2b48a6322}

| <b>Keyword</b>                | <b>Corresponding Legacy Flags and Groups</b> |
|-------------------------------|--|
| SYSTEM_PROFILE_KW_GENERAL     | PERF_PROFILE, EVENT_TRACE_FLAG_PROFILE       |
| SYSTEM_PROFILE_KW_PMC_PROFILE | PERF_PMC_PROFILE                             |

## System Registry Provider

Provides events relating to the registry.

GUID: SystemRegistryProviderGuid {16156bd9-fab4-4cfa-a232-89d1099058e3}

| <b>Keyword</b>                  | <b>Corresponding Legacy Flags and Groups</b> |
|---------------------------------|--|
| SYSTEM_REGISTRY_KW_GENERAL      | PERF_REGISTRY, EVENT_TRACE_FLAG_REGISTRY     |
| SYSTEM_REGISTRY_KW_HIVE         | PERF_REG_HIVE                                |
| SYSTEM_REGISTRY_KW_NOTIFICATION | PERF_REG_NOTIF                               |

## System Scheduler Provider

Provides events relating to the scheduler.

GUID: SystemSchedulerProviderGuid {599a2a76-4d91-4910-9ac7-7d33f2e97a6c}

| <b>Keyword</b>                      | <b>Corresponding Legacy Flags and Groups</b>     |
|-------------------------------------|--|
| SYSTEM_SCHEDULER_KW_XSCHEDULER      | PERF_XSCHEDULER                                  |
| SYSTEM_SCHEDULER_KW_DISPATCHER      | PERF_DISPATCHER,<br>EVENT_TRACE_FLAG_DISPATCHER  |
| SYSTEM_SCHEDULER_KW_KERNEL_QUEUE    | PERF_KERNEL_QUEUE                                |
| SYSTEM_SCHEDULER_KW_SHOULD_YIELD    | PERF_SHOULD_YIELD                                |
| SYSTEM_SCHEDULER_KW_ANTI_STARVATION | PERF_ANTI_STARVATION                             |
| SYSTEM_SCHEDULER_KW_LOAD_BALANCER   | PERF_LOAD_BALANCER                               |
| SYSTEM_SCHEDULER_KW_AFFINITY        | PERF_AFFINITY                                    |
| SYSTEM_SCHEDULER_KW_PRIORITY        | PERF_PRIORITY                                    |
| SYSTEM_SCHEDULER_KW_IDEAL_PROCESSOR | PERF_IDEAL_PROCESSOR                             |
| SYSTEM_SCHEDULER_KW_CONTEXT_SWITCH  | PERF_CONTEXT_SWITCH,<br>EVENT_TRACE_FLAG_CSWITCH |

## System Syscall Provider

Provides events with information about system calls.

GUID: SystemSyscallProviderGuid {434286f7-6f1b-45bb-b37e-95f623046c7c}

| <b>Keyword</b>            | <b>Corresponding Legacy Flags and Groups</b> |
|---------------------------|--|
| SYSTEM_SYSCALL_KW_GENERAL | PERF_SYSCALL, EVENT_TRACE_FLAG_SYSTEMCALL    |

# System Timer Provider

Provides events relating to timers in the kernel.

GUID: SystemTimerProviderGuid {4f061568-e215-499f-ab2e-eda0ae890a5b}

| Keyword                     | Corresponding Legacy Flags and Groups |
|-----------------------------|---------------------------------------|
| SYSTEM_TIMER_KW_GENERAL     | PERF_TIMER                            |
| SYSTEM_TIMER_KW_CLOCK_TIMER | PERF_CLOCK_TIMER                      |

## Remarks

This new enablement mechanism is in addition to the pre-existing methods for enabling these events. Any code that used to work, will continue to do so.

The events generated by the System Trace Provider are not changing due to this new feature. This means that the outputted events are not marked as being emitted from the individual system providers.

For more information on provider GUID and keyword definitions see [evntrace.h](#).

## See Also

[Configuring and Starting a SystemTraceProvider Session](#)

[evntrace.h header](#)

# Configuring and Starting an AutoLogger Session

Article • 10/18/2022 • 9 minutes to read

The AutoLogger event tracing session records events that occur early in the operating system boot process. Applications and device drivers can use the AutoLogger session to capture traces before the user logs in. Note that some device drivers, such as disk device drivers, are not loaded at the time the AutoLogger session begins.

The AutoLogger differs from the Global Logger in the following ways:

- You can specify one or more AutoLogger sessions (the Global Logger was a single session to which everyone logged events).
- The AutoLogger sends an enable notification to the providers when the session starts (the Global Logger did not send an enable notification to the providers, so the providers had to rely on other means to know if the Global Logger session was started in order to begin logging events).
- The AutoLogger does not support logging NT Kernel Logger events (see the **EnableFlags** member of [EVENT\\_TRACE\\_PROPERTIES](#)). To log NT Kernel Logger events, you must use the [Global Logger](#).

For more information on the Global Logger session, see [Configuring and Starting the Global Logger Session](#).

## ① Note

ETW supports the AutoLogger on Windows Vista and later. Use the [Global Logger](#) on earlier operating systems.

You use the registry to configure the AutoLogger session. Add the following registry key, if it is not already present:

```
HKEY_LOCAL_MACHINE  
  \SYSTEM  
    \CurrentControlSet  
      \Control  
        \WMI  
          \Autologger
```

Under the **Autologger** key create a key for each AutoLogger session that you want to configure as shown in the following example.

```
HKEY_LOCAL_MACHINE
  \SYSTEM
    \CurrentControlSet
      \Control
        \WMI
          \Autologger
            \Logger Session A
            \Logger Session B
            \Logger Session C
```

For each session, create a key for each provider that you want to enable to the session. Use the provider's GUID as the key.

```
HKEY_LOCAL_MACHINE
  \SYSTEM
    \CurrentControlSet
      \Control
        \WMI
          \Autologger
            \Logger Session A
              \{ProviderGuid1}
              \{ProviderGuid2}
            \Logger Session B
            \Logger Session C
```

The following table describes the values that you can define for each AutoLogger session. You must have administrator privileges to specify these registry values. The **Start** and **Guid** value are the only values required to start the AutoLogger session; all other values have default settings that are used if the value is not present in the registry. Typically, you should use the default values. If you specify a value that ETW cannot support, ETW will override the value.

| Value      | Type      | Description   |
|------------|-----------|---|
| BufferSize | REG_DWORD | The size of each buffer, in kilobytes. Should be less than one megabyte. ETW uses the size of physical memory to calculate this value.  |
| ClockType  | REG_DWORD | <p>The timer to use when logging the time stamp for each event.</p> <ul style="list-style-type: none"><li>• 1 = Performance counter value (high resolution)</li><li>• 2 = System timer</li><li>• 3 = CPU cycle counter</li></ul> <p>For a description of each clock type, see the <b>ClientContext</b> member of <a href="#">WNODE_HEADER</a>.</p> <p>The default value is 1 (performance counter value) on Windows Vista and later. Prior to Windows Vista, the default value is 2 (system timer).</p> |

| Value                      | Type      | Description   |
|----------------------------|-----------|---|
| DisableRealtimePersistence | REG_DWORD | <p>To disable real time persistence, set this value to 1. The default is 0 (enabled) for real time sessions.</p> <p>If real time persistence is enabled, real-time events that were not delivered by the time the computer was shutdown will be persisted. The events will then be delivered to the consumer the next time the consumer connects to the session.</p>  |
| FileCounter                | REG_DWORD | <p>Do not set or modify this value. This value is the serial number used to increment the log file name if <b>FileMax</b> is specified. If the value is not valid, 1 will be assumed.</p>   |
| FileName                   | REG_SZ    | <p>The fully qualified path of the log file. The path to this file must exist. The log file is a sequential log file. The path is limited to 1024 characters.</p> <p>If <b>FileName</b> is not specified, events are written to %SystemRoot%\System32\LogFiles\WMI\&lt;sessionname&gt;.etl.</p>   |
| FileMax                    | REG_DWORD | <p>The maximum number of instances of the log file that ETW creates. If the log file specified in <b>FileName</b> exists, ETW appends the <b>FileCounter</b> value to the file name. For example, if the default log file name is used, the form is %SystemRoot%\System32\LogFiles\WMI\&lt;sessionname&gt;.etl.NNNN. The first time the computer is started, the file name is &lt;sessionname&gt;.etl.0001, the second time the file name is &lt;sessionname&gt;.etl.0002, and so on. If <b>FileMax</b> is 3, on the fourth restart of the computer, ETW resets the counter to 1 and overwrites &lt;sessionname&gt;.etl.0001, if it exists.</p> <p>The maximum number of instances of the log file that are supported is 16.</p> <p>Do not use this feature with the <a href="#">EVENT_TRACE_FILE_MODE_NEWFILE</a> log file mode.</p> |
| FlushTimer                 | REG_DWORD | <p>How often, in seconds, the trace buffers are forcibly flushed. The minimum flush time is 1 second. This forced flush is in addition to the automatic flush that occurs when a buffer is full and when the trace session stops.</p> <p>For the case of a real-time logger, a value of zero (the default value) means that the flush time will be set to 1 second. A real-time logger is when <b>LogFileMode</b> is set to <a href="#">EVENT_TRACE_REAL_TIME_MODE</a>.</p> <p>The default value is 0. By default, buffers are flushed only when they are full.</p>   |
| Guid                       | REG_SZ    | <p>A string that contains a GUID that uniquely identifies the session. This value is required.</p>  |

| Value          | Type      | Description   |
|----------------|-----------|---|
| LogFileMode    | REG_DWORD | <p>Specify one or more log modes. For possible values, see <a href="#">Logging Mode Constants</a>. The default is <code>EVENT_TRACE_FILE_MODE_SEQUENTIAL</code>. Instead of writing to a log file, you can specify either <code>EVENT_TRACE_BUFFERING_MODE</code> or <code>EVENT_TRACE_REAL_TIME_MODE</code>.</p> <p>Specifying <code>EVENT_TRACE_BUFFERING_MODE</code> avoids the cost of flushing the contents of the session to disk when the file system becomes available.</p> <p>Note that using <code>EVENT_TRACE_BUFFERING_MODE</code> will cause the system to ignore the <code>MaximumBuffers</code> value, as the buffer size is instead the product of <code>MinimumBuffers</code> and <code>BufferSize</code>.</p> <p>AutoLogger sessions do not support the <code>EVENT_TRACE_FILE_MODE_NEWFILE</code> logging mode.</p> <p>If <code>EVENT_TRACE_FILE_MODE_APPEND</code> is specified, <code>BufferSize</code> must be explicitly provided and must be the same in both the logger and the file being appended.</p> |
| MaxFileSize    | REG_DWORD | <p>The maximum file size of the log file, in megabytes. The session is closed when the maximum size is reached, unless you are in circular log file mode. To specify no limit, set value to 0. The default is 100 MB, if not set. The behavior that occurs when the maximum file size is reached depends on the value of <code>LogFileMode</code>.</p>  |
| MaximumBuffers | REG_DWORD | <p>The maximum number of buffers to allocate. Typically, this value is the minimum number of buffers plus twenty. ETW uses the buffer size and the size of physical memory to calculate this value. This value must be greater than or equal to the value for <code>MinimumBuffers</code>.</p>  |
| MinimumBuffers | REG_DWORD | <p>The minimum number of buffers to allocate at startup. The minimum number of buffers that you can specify is two buffers per processor. For example, on a single processor computer, the minimum number of buffers is two.</p>  |
| Start          | REG_DWORD | <p>To have the AutoLogger session start the next time the computer is restarted, set this value to 1; otherwise, set this value to 0.</p>   |
| Status         | REG_DWORD | <p>The startup status of the AutoLogger. If the AutoLogger failed to start, the value of this key is the appropriate Win32 error code. If the AutoLogger successfully started, the value of this key is <code>ERROR_SUCCESS</code> (0).</p>   |
| Boot           | REG_DWORD | <p>This feature should not be used outside of debugging scenarios. If this registry key is set to 1, the autologger will be started earlier than normal during kernel initialization, allowing it to capture events during the initialization of many important kernel subsystems. However, enabling this option has a negative impact on boot times and imposes additional restrictions on the autologger. If this feature is enabled, the autologger session GUID must be populated, and many other autologger settings may not work.</p> <p>This key is supported on Windows Server 2022 and later.</p>  |

The following table describes the values that you can define for each provider that you want to enable to your session. You must have administrator privileges to specify these registry values. If you specify a value that ETW cannot support, ETW will override the value.

| Value           | Type      | Description   |
|-----------------|-----------|---|
| Enabled         | REG_DWORD | Determines if the provider is enabled. To enable the provider, set this value to 1. To disable the provider, set this value to 0. The default is 0.   |
| EnableFlags     | REG_DWORD | Provider-defined value that specifies the class of events for which the provider generates events. For details, see the <i>EnableFlags</i> parameter of the <a href="#">EnableTrace</a> function. Specify this value name if the provider does not support <b>MatchAnyKeyword</b> or <b>MatchAllKeyword</b> .   |
| EnableLevel     | REG_DWORD | Provider-defined value that specifies the level of detail included in the event. For example, you can use this value to indicate the severity level of the events (informational, warning, error) that the provider generates. For a list of predefined levels, see the <i>level</i> parameter of the <a href="#">EnableTraceEx</a> function.   |
| EnableProperty  | REG_DWORD | <p>Use this value to include one or more of the following items in the log file:</p> <ul style="list-style-type: none"> <li>• <b>EVENT_ENABLE_PROPERTY_SID</b> (0x00000001) = Include in the extended data the security identifier (SID) of the user.</li> <li>• <b>EVENT_ENABLE_PROPERTY_TS_ID</b> (0x00000002) = Include in the extended data the terminal session identifier.</li> <li>• <b>EVENT_ENABLE_PROPERTY_STACK_TRACE</b> (0x00000004) = Include in the extended data a call stack trace for events written using <a href="#">EventWrite</a>.</li> <li>• <b>EVENT_ENABLE_PROPERTY_IGNORE_KEYWORD_0</b> (0x00000010) = Filters out all events that do not have a non-zero keyword specified.</li> <li>• <b>EVENT_ENABLE_PROPERTY_PROVIDER_GROUP</b> (0x00000020) = Indicates that this call to <a href="#">EnableTraceEx2</a> should enable a <a href="#">Provider Group</a> rather than an individual Event Provider.</li> <li>• <b>EVENT_ENABLE_PROPERTY_PROCESS_START_KEY</b> (0x00000080) = Include the Process Start Key in the extended data.</li> <li>• <b>EVENT_ENABLE_PROPERTY_EVENT_KEY</b> (0x00000100) = Include the Event Key in the extended data.</li> <li>• <b>EVENT_ENABLE_PROPERTY_EXCLUDE_INPRIVATE</b> (0x00000200) = Filters out all events that are either marked as an InPrivate event or come from a process that is marked as InPrivate.</li> </ul> <p>For more information about these items, see the <b>EnableProperty</b> of the <a href="#">ENABLE_TRACE_PARAMETERS</a> structure.</p> |
| MatchAnyKeyword | REG_QWORD | Bitmask of keywords that determine the category of events that you want the provider to write. The provider writes the event if any of the event's keyword bits match any of the bits set in this mask. To specify that the provider write all events, set this value to zero. For an example, see the Remarks section of the <a href="#">EnableTraceEx</a> function.   |

| <b>Value</b>    | <b>Type</b> | <b>Description</b>  |
|-----------------|-------------|---|
| MatchAllKeyword | REG_QWORD   | This bitmask is optional. This mask further restricts the category of events that you want the provider to write. If the event's keyword meets the <i>MatchAnyKeyword</i> condition, the provider will write the event only if all of the bits in this mask exist in the event's keyword. This mask is not used if <i>MatchAnyKeyword</i> is zero. For an example, see the Remarks section of the <a href="#">EnableTraceEx</a> function. |

After the registry has been modified, the AutoLogger session is started the next time the computer is restarted. The AutoLogger session calls the [EnableTraceEx](#) function to enable the providers.

The AutoLogger sessions increase the system boot time and should be used sparingly. Services that want to capture information during the boot process should consider adding controller logic to itself instead of using the AutoLogger session.

To stop an AutoLogger session, call the [ControlTrace](#) function. The session name that you pass to the function is the name of the registry key that you used to define the session in the registry.

For details on starting an event tracing session, see [Configuring and Starting an Event Tracing Session](#).

For details on starting a private logger session, see [Configuring and Starting a Private Logger Session](#).

For details on starting an NT Kernel Logger session, see [Configuring and Starting the NT Kernel Logger Session](#).

## Related topics

[Configuring and Starting a Private Logger Session](#)

[Configuring and Starting a SystemTraceProvider Session](#)

[Configuring and Starting an Event Tracing Session](#)

[Configuring and Starting the NT Kernel Logger Session](#)

[EnableTraceEx2](#)

[ENABLE\\_TRACE\\_PARAMETERS](#)

[EVENT\\_FILTER\\_DESCRIPTOR](#)

[PAYLOAD\\_FILTER\\_PREDICATE](#)

[TdhAggregatePayloadFilters](#)

[TdhCreatePayloadFilter](#)

## Updating an Event Tracing Session

# Configuring and Starting the Global Logger Session

Article • 08/19/2021 • 5 minutes to read

The Global Logger event tracing session records events that occur early in the operating system boot process. Applications and device drivers can use the Global Logger session to capture traces before the user logs in. Note that some device drivers, such as disk device drivers, are not loaded at the time the Global Logger session begins.

## ⓘ Note

If you are creating a Global Logger session on Windows Vista, you should consider creating an [AutoLogger session](#) instead.

You use the registry to configure the Global Logger session. Add the **GlobalLogger** key to the following registry key, if it is not already present:

```
HKEY_LOCAL_MACHINE  
  \SYSTEM  
    \CurrentControlSet  
      \Control  
        \WMI
```

The following table describes the values that you can define for the **GlobalLogger** key. You must have administrator privileges to specify these registry values. The registry values affect all providers that log events to the Global Logger session. The **Start** value is the only value required to start the Global Logger session; all other values have default settings that are used if the value is not present in the registry. Typically, you should use the default values. If you specify a value that ETW cannot support, ETW will override the value.

| Value      | Type      | Description  |
|------------|-----------|--|
| Start      | REG_DWORD | Set this value to 1 (on) to start the Global Logger session the next time the system starts. To stop the session from starting, set this value to 0 (off). |
| BufferSize | REG_DWORD | The size of each buffer, in kilobytes. This value should be less than one megabyte. ETW uses the size of physical memory to calculate this value.          |

| Value             | Type       | Description  |
|-------------------|------------|--|
| ClockType         | REG_DWORD  | <p>The timer to use when logging the time stamp for each event.</p> <ul style="list-style-type: none"> <li>• 1 = Performance counter value (high resolution)</li> <li>• 2 = System timer</li> <li>• 3 = CPU cycle counter</li> </ul>   |
|                   |            | <p>For a description of each clock type, see the <b>ClientContext</b> member of <a href="#">WNODE_HEADER</a>.</p>  |
|                   |            | <p>The default value is 1 (performance counter value) on Windows Vista and later. Prior to Windows Vista, the default value is 2 (system timer).</p>   |
| EnableKernelFlags | REG_BINARY | <p>Use this value to enable one or more kernel providers. If you enable kernel providers, the Global Logger session will rename itself to NT Kernel Logger when it starts. For possible values, see the <b>EnableFlags</b> member of <a href="#">EVENT_TRACE_PROPERTIES</a>.</p>   |
| FileCounter       | REG_DWORD  | <p>The number of event trace log files generated by Global Logger sessions. The system increments this value until it reaches the value of <b>FileMax</b>. Then, it resets the value to 0. This counter prevents the system from overwriting a Global Logger trace log file.</p>   |
| FileMax           | REG_DWORD  | <p>The maximum number of event trace log files permitted on the system. When the number of trace logs reaches the specified maximum, the system begins to overwrite the logs, beginning with the oldest.</p> <p>If the log file specified in <b>FileName</b> exists, ETW appends the <b>FileCounter</b> value to the file name. For example, if the default log file name is used, the form is %SystemRoot%\System32\LogFiles\WMI\GlobalLogger.etl.NNNN. The default value is 0, meaning that there is no maximum.</p> |
| FileName          | REG_SZ     | <p>Fully qualified path of the log file. The path to this file must exist. The log file is a sequential log file. Note that all providers writing events to the Global Logger session write events to this log file. The path is limited to 1024 characters. If <b>FileName</b> is not specified, events are written to %SystemRoot%\System32\LogFiles\WMI\GlobalLogger.etl. <b>Prior to Windows Vista:</b> The default file is %SystemRoot%\System32\LogFiles\WMI\Trace.log.</p>                                      |

| Value          | Type      | Description  |
|----------------|-----------|--|
| FlushTimer     | REG_DWORD | <p>How often, in seconds, the trace buffers are forcibly flushed. The minimum flush time is 1 second. This forced flush is in addition to the automatic flush that occurs when a buffer is full and when the trace session stops.</p> <p>For the case of a real-time logger, a value of zero (the default value) means that the flush time will be set to 1 second. A real-time logger is when <b>LogFileMode</b> is set to <b>EVENT_TRACE_REAL_TIME_MODE</b>.</p> <p>The default value is 0. By default, buffers are flushed only when they are full.</p> |
| LogFileMode    | REG_DWORD | Specifies log session options. For values, see <a href="#">Logging Mode Constants</a> . This values is supported on Windows Vista and later.   |
| MaximumBuffers | REG_DWORD | The maximum number of buffers to allocate. Typically, this value is the minimum number of buffers plus twenty. ETW uses the buffer size and the size of physical memory to calculate this value. This value must be greater than or equal to the value for <b>MinimumBuffers</b> .   |
| MaxFileSize    | REG_DWORD | The maximum size, in megabytes, of the event trace log file. By default, there is no maximum file size.  |
| MinimumBuffers | REG_DWORD | The minimum number of buffers to allocate when the Global Logger session starts. The minimum number of buffers that you can specify is two buffers per processor. For example, on a single processor computer, the minimum number of buffers is two. The default value on a single-processor system is 0x3.  |
| Status         | REG_DWORD | The startup status of the Global Logger. If the Global Logger failed to start, the value of this key is the appropriate Win32 error code. If the Global Logger successfully started, the value of this key is ERROR_SUCCESS (0).   |

After the registry has been modified and the computer restarted, the Global Logger session starts automatically and is used like any other session with one exception: You use the **WMI\_GLOBAL\_LOGGER\_ID** constant handle (defined in Wmistr.h) to reference the Global Logger session. This constant may be used as an argument to any event tracing function that accepts a session handle. In functions that accept a session name, use **GLOBAL\_LOGGER\_NAME**.

The Global Logger controller does not call the **EnableTrace** function to enable providers. The provider is responsible for determining if the Global Logger session is started and then enabling itself.

To determine if the Global Logger session is started, you can call the [ControlTrace](#) function, setting *SessionHandle* to WMI\_GLOBAL\_LOGGER\_ID and *ControlCode* to **EVENT\_TRACE\_CONTROL\_QUERY**. If the **ControlTrace** call is successful, the Global Logger session exists and the provider can enable itself and log events to the Global Logger session (the **ControlTrace** function returns **ERROR\_WMI\_INSTANCE\_NOT\_FOUND** if the Global Logger is not active).

Typically, the controller is responsible for passing the enable flags and level to the provider when it enables the provider, but because the Global Logger controller does not enable the provider, it is the provider's responsibility to pass this information to itself, if needed.

The Global Logger session is a limited resource and should be used sparingly. Services that want to capture information during the boot process should consider adding controller logic to itself instead of using the Global Logger session.

For details on starting an event tracing session, see [Configuring and Starting an Event Tracing Session](#).

For details on starting a private logger session, see [Configuring and Starting a Private Logger Session](#).

For details on starting an NT Kernel Logger session, see [Configuring and Starting the NT Kernel Logger Session](#).

# Configuring and Starting a Private Logger Session

Article • 06/01/2022 • 2 minutes to read

A private event tracing session is a user-mode event tracing session that runs in the same process as its event trace providers—the private session and the providers that it enables must all be in the same process. The benefit of using a private session is that the private session does not count against the maximum of 64 event tracing sessions executing simultaneously.

Configuring and starting a private session is similar to starting a normal event trace session. The difference is that `Wnode.Guid` member of the [EVENT\\_TRACE\\_PROPERTIES](#) structure must contain the **GUID** of the provider, not the session, and the provider must have already registered the GUID. Note that if you also set the `EVENT_TRACE_PRIVATE_IN_PROC` logging mode, you can use a different **GUID** for the session and provider. For details on starting a normal event trace session, see [Configuring and Starting an Event Tracing Session](#).

Note that you cannot start, stop, or flush a private trace session from `DllMain`; you should do so in the initialization and finalization routines of the DLL.

From Windows 8.1 to Windows 10, version 1607, event payload, scope, and stack walk filters can be used by the [EnableTraceEx2](#) function and the [ENABLE\\_TRACE\\_PARAMETERS](#) and [EVENT\\_FILTER\\_DESCRIPTOR](#) structures to filter on specific conditions in a logger session. For more information on event payload filters, see the [TdhCreatePayloadFilter](#), and [TdhAggregatePayloadFilters](#) functions and the [ENABLE\\_TRACE\\_PARAMETERS](#), [EVENT\\_FILTER\\_DESCRIPTOR](#), and [PAYLOAD\\_FILTER\\_PREDICATE](#) structures.

Starting with Windows 10, version 1703, low privilege users can now start a private logger session in processes they started. The provider no longer needs to be registered prior to enabling or starting the private session, meaning the provider is "pre-enabled" similar to how non-private session providers are. There is a limit of 8 system wide private loggers to an individual process. For increased performance in cross process scenarios, it's recommended to use filtering for session APIs (including [ControlTrace](#), [QueryTrace](#), [StartTrace](#), and [StopTrace](#)) when starting a system wide private logger. Note that the same filters should be passed to all session APIs. For more information about filters, see [EVENT\\_TRACE\\_PROPERTIES\\_V2](#).

For details on starting an event tracing session, see [Configuring and Starting an Event Tracing Session](#).

For details on starting an NT Kernel Logger session, see [Configuring and Starting the NT Kernel Logger Session](#).

For details on starting a Global Logger session, see [Configuring and Starting a Global Logger Session](#).

For details on starting an AutoLogger session, see [Configuring and Starting an AutoLogger Session](#).

## Related topics

[Configuring and Starting a SystemTraceProvider Session](#)

[Configuring and Starting an AutoLogger Session](#)

[Configuring and Starting an Event Tracing Session](#)

[Configuring and Starting the NT Kernel Logger Session](#)

[EnableTraceEx2](#)

[ENABLE\\_TRACE\\_PARAMETERS](#)

[EVENT\\_TRACE\\_PROPERTIES](#)

[EVENT\\_TRACE\\_PROPERTIES\\_V2](#)

[EVENT\\_FILTER\\_DESCRIPTOR](#)

[PAYLOAD\\_FILTER\\_PREDICATE](#)

[TdhAggregatePayloadFilters](#)

[TdhCreatePayloadFilter](#)

[Updating an Event Tracing Session](#)

# Updating an Event Tracing Session

Article • 01/07/2021 • 2 minutes to read

To update the properties of an event tracing session, call the [ControlTrace](#) function using the **EVENT\_TRACE\_CONTROL\_UPDATE** control code. You can specify the session to update using a session handle obtained from an earlier call to [StartTrace](#), or a session name. The properties of the event tracing session are updated using the values specified in the [EVENT\\_TRACE\\_PROPERTIES](#) structure. You can update only a subset of the properties. For a list of the properties you can update, see the *Properties* parameter of the [ControlTrace](#) function.

If the [ControlTrace](#) call is successful, the [EVENT\\_TRACE\\_PROPERTIES](#) structure is updated to reflect the new property values. The structure will also contain the new run statistics for the event tracing session.

The following example shows how to update the kernel event tracing session. The example queries for the current property values and updates the structure before updating the session.

C++

```
#include <windows.h>
#include <stdio.h>
#include <wmistr.h>
#include <evntrace.h>

#define MAX_SESSION_NAME_LEN 1024
#define MAX_LOGFILE_PATH_LEN 1024

void wmain(void)
{
    ULONG status = ERROR_SUCCESS;
    EVENT_TRACE_PROPERTIES* pSessionProperties = NULL;
    ULONG BufferSize = 0;

    // Allocate memory for the session properties. The memory must
    // be large enough to include the log file name and session name.
    // This example specifies the maximum size for the session name
    // and log file name.

    BufferSize = sizeof(EVENT_TRACE_PROPERTIES) +
        (MAX_SESSION_NAME_LEN * sizeof(WCHAR)) +
        (MAX_LOGFILE_PATH_LEN * sizeof(WCHAR));

    pSessionProperties = (EVENT_TRACE_PROPERTIES*) malloc(BufferSize);
    if (NULL == pSessionProperties)
    {
        wprintf(L"Unable to allocate %d bytes for properties structure.\n",
    }
```

```

BufferSize);
    goto cleanup;
}

ZeroMemory(pSessionProperties, BufferSize);
pSessionProperties->Wnode.BufferSize = BufferSize;

// Query for the kernel trace session's current property values.

status = ControlTrace((TRACEHANDLE)NULL, KERNEL_LOGGER_NAME,
pSessionProperties, EVENT_TRACE_CONTROL_QUERY);

if (ERROR_SUCCESS != status)
{
    if (ERROR_WMI_INSTANCE_NOT_FOUND == status)
    {
        wprintf(L"The Kernel Logger is not running.\n");
    }
    else
    {
        wprintf(L"ControlTrace(query) failed with %lu\n", status);
    }

    goto cleanup;
}

// Update the enable flags to also enable the Process and Thread
providers.

pSessionProperties->LogFileNameOffset = 0; // Zero tells ETW not to
update the file name
pSessionProperties->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
pSessionProperties->EnableFlags |= EVENT_TRACE_FLAG_PROCESS |
EVENT_TRACE_FLAG_THREAD;

// Update the session's properties.

status = ControlTrace((TRACEHANDLE)NULL, KERNEL_LOGGER_NAME,
pSessionProperties, EVENT_TRACE_CONTROL_UPDATE);
if (ERROR_SUCCESS != status)
{
    wprintf(L"ControlTrace(update) failed with %lu.\n", status);
    goto cleanup;
}

wprintf(L"\nUpdated session");

cleanup:

if (pSessionProperties)
{
    free(pSessionProperties);
    pSessionProperties = NULL;
}
}

```

## Related topics

[Configuring and Starting a Private Logger Session](#)

[Configuring and Starting a SystemTraceProvider Session](#)

[Configuring and Starting an AutoLogger Session](#)

[Configuring and Starting an Event Tracing Session](#)

[Configuring and Starting the NT Kernel Logger Session](#)

# Retrieving Additional Event Tracing Data

Article • 01/07/2021 • 2 minutes to read

Once you have begun an event tracing session, you can use [TraceSetInformation](#) to instruct the system to return additional event tracing data. The additional information will be placed in the extended data section of the relevant event trace.

The following procedure describes how to use the [TraceSetInformation](#) function to retrieve additional data from an event tracing session.

## To retrieve additional event tracing data

1. Start your session with a call to [StartTrace](#).

For more information, see [Configuring and Starting an Event Tracing Session](#).

2. Call [TraceSetInformation](#) to set additional event tracing data.

use the [EVENT\\_INFO\\_CLASS](#) enumeration in the *ClassInformation* parameter to describe the additional information you wish to retrieve. The following example describes how to call [TraceSetInformation](#), using the session handle returned from the call to [StartTrace](#), and the [TraceProviderBinaryTracking](#) value from [EVENT\\_INFO\\_CLASS](#).

C++

```
BOOLEAN enabled = TRUE;
Win32Error error = TraceSetInformation(
    m_sessionHandle,
    TraceProviderBinaryTracking,
    &enabled,
    sizeof(enabled));
```

3. Alternately, you can use [TraceQueryInformation](#) to retrieve information about the current event tracing session settings.

Like [TraceSetInformation](#), [TraceQueryInformation](#) uses the [EVENT\\_INFO\\_CLASS](#) enumeration to describe what information to retrieve from the system.

# Providing Events

Article • 01/07/2021 • 2 minutes to read

Providers are applications that contain event tracing instrumentation. After a provider registers itself, a controller can then enable or disable event tracing in the provider. The provider defines its interpretation of being enabled or disabled. Generally, an enabled provider generates events, and a disabled provider does not. This lets you add event tracing to your application without requiring that it generate events all the time.

This section shows you how to:

- [Write events](#)
- [Write related events](#)
- [Publish your event schema to share with consumers](#)

For information about controlling event tracing sessions, see [Controlling Event Tracing Sessions](#). For information about consuming events from an event trace provider, see [Consuming Events](#).

# Writing Events

Article • 01/07/2021 • 2 minutes to read

How you write events to a trace session is determined by the type of your provider.

For [MOF](#) providers, see [Writing MOF \(Classic\) Events](#).

For [WPP](#) providers, see [Writing WPP Events](#).

For [manifest-based](#) providers, see [Writing Manifest-based Events](#).

For [TraceLogging](#) providers, see [Using TraceLogging](#).

# Writing Manifest-based Events

Article • 06/16/2021 • 5 minutes to read

Before you can write events to a trace session, you must register your provider. Registering a provider tells ETW that your provider is ready to write events to a trace session. A process can register up to 1,024 provider GUIDs; however, you should limit the number of providers that your process registers to one or two.

**Prior to Windows Vista:** There is no limit to the number of providers that a process can register.

To register a manifest-based provider, call the [EventRegister](#) function. The function registers the provider's GUID and identifies an optional callback that ETW calls when a controller enables or disables the provider.

Before the provider exits, call the [EventUnregister](#) function to remove the provider's registration from ETW. The [EventRegister](#) function returns the registration handle that you pass to the [EventUnregister](#) function.

Manifest-based providers do not have to implement an [EnableCallback](#) function to receive notifications when a session enables or disables the provider. The callback is optional and is used for informational purposes; you do not need to specify or implement the callback when registering the provider. A manifest-based provider can simply write events and ETW will decide if the event gets logged to a trace session. If an event requires that you perform extensive work to generate the event's data, you may want to call the [EventEnabled](#) or [EventProviderEnabled](#) function first to verify that the event will be written to a session before performing the work.

Typically, you would implement the callback if your provider requires that the controller pass provider-defined filter data (see the *FilterData* parameter of [EnableCallback](#)) to the provider, or the provider uses the context information that it specified when it registered itself (see the *CallbackContext* parameter of [EventRegister](#)).

Manifest-based providers call the [EventWrite](#) or [EventWriteString](#) function to write events to a session. If your event data is a string, or if you do not define a manifest for your provider and your event data is a single string, call the [EventWriteString](#) function to write the event. For event data that contains numeric or complex data types, call the [EventWrite](#) function to log the event.

The following example shows how to prepare the event data to be written using the [EventWrite](#) function. The example references the events defined in [Publishing Your Event Schema for a Manifest-based Provider](#).

C++

```
#include <windows.h>
#include <stdio.h>
#include <evntprov.h>
#include "provider.h" // Generated from manifest

#define SUNDAY      0X1
#define MONDAY      0X2
#define TUESDAY     0X4
#define WEDNESDAY   0X8
#define THURSDAY    0X10
#define FRIDAY      0X20
#define SATURDAY    0X40

enum TRANSFER_TYPE {
    Download = 1,
    Upload,
    UploadReply
};

#define MAX_NAMEDVALUES          5 // Maximum array size
#define MAX_PAYLOAD_DESCRIPTOROS 9 + (2 * MAX_NAMEDVALUES)

typedef struct _namedvalue {
    LPWSTR name;
    USHORT value;
} NAMEDVALUE, *PNAMEDVALUE;

void wmain(void)
{
    DWORD status = ERROR_SUCCESS;
    REGHANDLE RegistrationHandle = NULL;
    EVENT_DATA_DESCRIPTOR Descriptors[MAX_PAYLOAD_DESCRIPTOROS];
    DWORD i = 0;

    // Data to load into event descriptors

    USHORT Scores[3] = {45, 63, 21};
    ULONG pImage = (ULONG)&Scores;
    DWORD TransferType = Upload;
    DWORD Day = MONDAY | TUESDAY;
    LPWSTR Path = L"c:\\path\\\\folder\\file.ext";
    BYTE Cert[11] = {0x2, 0x4, 0x8, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x0, 0x1};
    PBYTE Guid = (PBYTE) &ProviderGuid;
    USHORT ArraySize = MAX_NAMEDVALUES;
    BOOL IsLocal = TRUE;
    NAMEDVALUE NamedValues[MAX_NAMEDVALUES] = {
        {L"Bill", 1},
        {L"Bob", 2},
        {L"William", 3},
        {L"Robert", 4},
        {L"", 5}
    };
}
```

```

};

status = EventRegister(
    &ProviderGuid,          // GUID that identifies the provider
    NULL,                  // Callback not used
    NULL,                  // Context noot used
    &RegistrationHandle // Used when calling EventWrite and
EventUnregister
);

if (ERROR_SUCCESS != status)
{
    wprintf(L"EventRegister failed with %lu\n", status);
    goto cleanup;
}

// Load the array of data descriptors for the TransferEvent event.
// Add the data to the array in the order of the <data> elements
// defined in the event's template.

EventDataDescCreate(&Descriptors[i++], &pImage, sizeof(ULONG));
EventDataDescCreate(&Descriptors[i++], Scores, sizeof(Scores));
EventDataDescCreate(&Descriptors[i++], Guid, sizeof(GUID));
EventDataDescCreate(&Descriptors[i++], Cert, sizeof(Cert));
EventDataDescCreate(&Descriptors[i++], &IsLocal, sizeof(BOOL));
EventDataDescCreate(&Descriptors[i++], Path, (ULONG)(wcslen(Path) + 1) *
sizeof(WCHAR));
EventDataDescCreate(&Descriptors[i++], &ArraySize, sizeof(USHORT));

// If your event contains a structure, you should write each member
// of the structure separately. If the structure contained integral data
types
// such as DWORDs and the data types were aligned on an 8-byte boundary,
you
// could use the following call to write the structure, however, you are
// encouraged to write the members separately.
//
// EventDataDescCreate(&EvtData, struct, sizeof(struct));
//
// Because the array of structures in this example contains both strings
// and numbers, you must write each member of the structure separately.

for (int j = 0; j < MAX_NAMEDVALUES; j++)
{
    EventDataDescCreate(&Descriptors[i++], NamedValues[j].name, (ULONG)
(wcslen(NamedValues[j].name)+1) * sizeof(WCHAR) );
    EventDataDescCreate(&Descriptors[i++], &(NamedValues[j].value),
sizeof(USHORT) );
}

EventDataDescCreate(&Descriptors[i++], &Day, sizeof(DWORD));
EventDataDescCreate(&Descriptors[i++], &TransferType, sizeof(DWORD));

// Write the event. You do not have to verify if your provider is
enabled before

```

```

        // writing the event. ETW will write the event to any session that
enabled
        // the provider. If no session enabled the provider, the event is not
        // written. If you need to perform extra work to write an event that you
        // would not otherwise do, you may want to call the EventEnabled
function
        // before performing the extra work. The EventEnabled function tells you
if a
        // session has enabled your provider, so you know if you need to perform
the
        // extra work or not.

    status = EventWrite(
        RegistrationHandle,                      // From EventRegister
        &TransferEvent,                          // EVENT_DESCRIPTOR generated from
the manifest
        (ULONG)MAX_PAYLOAD_DESCRIPTORS,          // Size of the array of
EVENT_DATA_DESCRIPTORs
        &Descriptors[0]                           // Array of descriptors that
contain the event data
    );

    if (status != ERROR_SUCCESS)
    {
        wprintf(L"EventWrite failed with 0x%x", status);
    }

cleanup:

    EventUnregister(RegistrationHandle);
}

```

When you compile the manifest (see [Compiling an Instrumentation Manifest](#)) that the example above uses, it creates the following header file (referenced in the example above).

C++

```

//****************************************************************************
/* This is an include file generated by Message Compiler.      */
/*
/* Copyright (c) Microsoft Corporation. All Rights Reserved.   */
//****************************************************************************
#pragma once
//+
// Provider Microsoft-Windows-ETWProvider Event Count 1
//+
EXTERN_C __declspec(selectany) const GUID ProviderGuid = {0xd8909c24,
0x5be9, 0x4502, {0x98, 0xca, 0xab, 0xb, 0xdc, 0x24, 0x89, 0x9d}};
//
// Keyword
//
#define READ_KEYWORD 0x1

```

```
#define WRITE_KEYWORD 0x2
#define LOCAL_KEYWORD 0x4
#define REMOTE_KEYWORD 0x8

//
// Event Descriptors
//
EXTERN_C __declspec(selectany) const EVENT_DESCRIPTOR TransferEvent = {0x1,
0x0, 0x0, 0x4, 0x0, 0x0, 0x5};
#define TransferEvent_value 0x1
#define MSG_Provider_Name 0x90000001L
#define MSG_Event_WhenToTransfer 0xB0000001L
#define MSG_Map_Download 0xD0000001L
#define MSG_Map_Upload 0xD0000002L
#define MSG_Map_UploadReply 0xD0000003L
#define MSG_Map_Sunday 0xF0000001L
#define MSG_Map_Monday 0xF0000002L
#define MSG_Map_Tuesday 0xF0000003L
#define MSG_Map_Wednesday 0xF0000004L
#define MSG_Map_Thursday 0xF0000005L
#define MSG_Map_Friday 0xF0000006L
#define MSG_Map_Saturday 0xF0000007L
```

# Writing MOF (Classic) Events

Article • 06/16/2021 • 9 minutes to read

Before you can write events to a trace session, you must register your provider. Registering a provider tells ETW that your provider is ready to write events to a trace session. A process can register up to 1,024 provider GUIDs; however, you should limit the number of providers that your process registers to one or two.

**Prior to Windows Vista:** There is no limit to the number of providers that a process can register.

To register a classic provider, call the [RegisterTraceGuids](#) function. The function registers the provider's GUID, event trace class GUIDs, and identifies the callback that ETW calls when a controller enables or disables the provider.

If the provider calls the [TraceEvent](#) function to log events, you do not need to include the array of class GUIDs (can be **NULL**) when calling the [RegisterTraceGuids](#) function. You only need to include the array if the provider calls the [TraceEventInstance](#) function to log events.

**Windows XP and Windows 2000:** You always need to include the array of class GUIDs (cannot be **NULL**).

After a provider registers itself and is enabled by the controller, the provider can log events to the controller's trace session.

Before the provider exits, call the [UnregisterTraceGuids](#) function to remove the provider's registration from ETW. The [RegisterTraceGuids](#) function returns the registration handle that you pass to the [UnregisterTraceGuids](#) function.

If your provider logs events to the Global Logger session only, you do not have to register your provider with ETW because the Global Logger controller does not enable or disable providers. For details, see [Configuring and Starting the Global Logger Session](#).

All [classic](#) providers (except those that trace events to the Global Logger session) must implement the [ControlCallback](#) function. The provider uses the information in the callback to determine if it is enabled or disabled and which events it should write.

The provider specifies the name of the callback function when it calls the [RegisterTraceGuids](#) function to register itself. ETW calls the callback function when the controller calls the [EnableTrace](#) function to enable or disable the provider.

In your [ControlCallback](#) implementation, you must call the [GetTraceLoggerHandle](#) function to retrieve the session handle; you use the session handle when calling the [TraceEvent](#) function. You only need to call the [GetTraceEnableFlags](#) function or the [GetTraceEnableLevel](#) function in your [ControlCallback](#) implementation if your provider uses the enable level or enable flags.

A provider can log trace events to only one session, but there is nothing to prevent multiple controllers from enabling a single provider. To prevent another controller from redirecting your trace events to its session, you may want to add logic to your [ControlCallback](#) implementation to compare session handles and ignore enable requests from other controllers.

To log events, [classic](#) providers call the [TraceEvent](#) function. An event consists of the [EVENT\\_TRACE\\_HEADER](#) structure and any event-specific data that is appended to the header.

The header must contain the following information:

- The **Size** member must contain the total number of bytes to be recorded for the event (including the size of the [EVENT\\_TRACE\\_HEADER](#) structure and of any event-specific data that is appended to the header).
- The **Guid** member must contain the class GUID of the event (or the **GuidPtr** member must contain a pointer to the class GUID).

**Windows XP and Windows 2000:** The class GUID must have been registered previously using the [RegisterTraceGuids](#) function.

- The **Flags** member must contain the [WNODE\\_FLAG\\_TRACED\\_GUID](#) flag. If you specify the class GUID using the **GuidPtr** member, also add the [WNODE\\_FLAG\\_USE\\_GUID\\_PTR](#) flag.
- The **Class.Type** member must contain the event type, if you use MOF to publish the layout of your event data.
- The **Class.Version** member must contain the event version, if you use MOF to publish the layout of your event data. The version is used to distinguish between revisions to the event data. Set the initial version number to 0.

If you write both the provider and the consumer, you can use a structure to populate the event-specific data that is appended to the header. However, if you use MOF to publish the event-specific data so that any consumer can process the event, you should not use a structure to append the event-specific data to the header. This is because the compiler may add extra bytes to the event-specific data for byte alignment purposes.

Because the MOF definition does not account for the extra bytes, the consumer may retrieve data that is not valid.

You should either allocate a block of memory for the event and copy each event data item to the memory, or create a structure that includes an array of **MOF\_FIELD** structures; most applications will create a structure that includes an array of **MOF\_FIELD** structures. Make sure that **Header.Size** reflects the actual number of **MOF\_FIELD** structures that are actually set before logging the event. For example, if the event contains three data fields, set **Header.Size** to `sizeof(EVENT_TRACE_HEADER) + (sizeof(MOF_FIELD) * 3)`.

For information on tracing related events, see [Writing Related Events in an End-to-End Scenario](#).

The following example shows how to call the **TraceEvent** function to log events. The example references the events defined in [Publishing Your Event Schema for a Classic Provider](#).

C++

```
#include <windows.h>
#include <stdio.h>
#include <wmistr.h>
#include <evntrace.h>

// GUID that identifies the category of events that the provider can log.
// The GUID is also used in the event MOF class.
// Remember to change this GUID if you copy and paste this example.

// {B49D5931-AD85-4070-B1B1-3F81F1532875}
static const GUID MyCategoryGuid =
{ 0xb49d5931, 0xad85, 0x4070, { 0xb1, 0xb1, 0x3f, 0x81, 0xf1, 0x53, 0x28,
0x75 } };

// GUID that identifies the provider that you are registering.
// The GUID is also used in the provider MOF class.
// Remember to change this GUID if you copy and paste this example.

// {7C214FB1-9CAC-4b8d-BAED-7BF48BF63BB3}
static const GUID ProviderGuid =
{ 0x7c214fb1, 0x9cac, 0x4b8d, { 0xba, 0xed, 0x7b, 0xf4, 0x8b, 0xf6, 0x3b,
0xb3 } };

// Identifies the event type within the MyCategoryGuid category
// of events to be logged. This is the same value as the EventType
// qualifier that is defined in the event type MOF class for one of
// the MyCategoryGuid category of events.

#define MY_EVENT_TYPE 1
```

```

// Event passed to TraceEvent

typedef struct _event
{
    EVENT_TRACE_HEADER Header;
    MOF_FIELD Data[MAX_MOF_FIELDS]; // Event-specific data
} MY_EVENT, *PMY_EVENT;

#define MAX_INDICES          3
#define MAX_SIGNATURE_LEN    32
#define EVENT_DATA_FIELDS_CNT 6

// Application data to be traced for Version 1 of the MOF class.

typedef struct _evtdatas
{
    LONG Cost;
    DWORD Indices[MAX_INDICES];
    WCHAR Signature[MAX_SIGNATURE_LEN];
    BOOL IsComplete;
    GUID ID;
    DWORD Size;
}EVENT_DATA, *PEVENT_DATA;

// GUID used as the value for EVENT_DATA.ID.

// {25BAEDA9-C81A-4889-8764-184FE56750F2}
static const GUID tempID =
{ 0x25baeda9, 0xc81a, 0x4889, { 0x87, 0x64, 0x18, 0x4f, 0xe5, 0x67, 0x50,
0xf2 } };

// Global variables to store tracing state information.

TRACEHANDLE g_SessionHandle = 0; // The handle to the session that enabled
the provider.

ULONG g_EnableFlags = 0; // Determines which class of events to log.
UCHAR g_EnableLevel = 0; // Determines the severity of events to log.
BOOL g_TraceOn = FALSE; // Used by the provider to determine whether it
should log events.

ULONG WINAPI ControlCallback(WMIDPREQUESTCODE RequestCode, PVOID Context,
ULONG* Reserved, PVOID Header);

// For this example to log the event, run the session example
// to enable the provider before running this example.

void wmain(void)
{
    ULONG status = ERROR_SUCCESS;
    EVENT_DATA EventData;
    MY_EVENT MyEvent;
    TRACEHANDLE RegistrationHandle = 0;
    TRACE_GUID_REGISTRATION EventClassGuids[] = {
        (LPGUID)&MyCategoryGuid, NULL
    };
}

```

```

// Register the provider and specify the control callback function
// that receives the enable/disable notifications.

status = RegisterTraceGuids(
    (WMIDPREQUEST)ControlCallback,
    NULL,
    (LPGUID)&ProviderGuid,
    sizeof(EventClassGuids)/sizeof(TRACE_GUID_REGISTRATION),
    EventClassGuids,
    NULL,
    NULL,
    &RegistrationHandle
);

if (ERROR_SUCCESS != status)
{
    wprintf(L"RegisterTraceGuids failed with %lu\n", status);
    goto cleanup;
}

// Set the event-specific data.

EventData.Cost = 32;
EventData.ID = tempID;
EventData.Indices[0] = 4;
EventData.Indices[1] = 5;
EventData.Indices[2] = 6;
EventData.IsComplete = TRUE;
wcscpy_s(EventData.Signature, MAX_SIGNATURE_LEN, L"Signature");
EventData.Size = 1024;

// Log the event if the provider is enabled and the session
// passed a severity level value >= TRACE_LEVEL_ERROR (or zero).
// If your provider generates more than one class of events, you
// would also need to check g_EnableFlags.

if (g_TraceOn && (0 == g_EnableLevel || TRACE_LEVEL_ERROR <=
g_EnableLevel))
{
    // Initialize the event data structure.

    ZeroMemory(&MyEvent, sizeof(MY_EVENT));
    MyEvent.Header.Size = sizeof(EVENT_TRACE_HEADER) +
(sizeof(MOF_FIELD) * EVENT_DATA_FIELDS_CNT);
    MyEvent.Header.Flags = WNODE_FLAG_TRACED_GUID |
WNODE_FLAG_USE_MOF_PTR;
    MyEvent.Header.Guid = MyCategoryGuid;
    MyEvent.Header.Class.Type = MY_EVENT_TYPE;
    MyEvent.Header.Class.Version = 1;
    MyEvent.Header.Class.Level = g_EnableLevel;

    // Load the event data. You can also use the DEFINE_TRACE_MOF_FIELD
    // macro defined in evntrace.h to set the MOF_FIELD members. For
example,

```

```

        // DEFINE_TRACE_MOF_FIELD(&EventData.Data[0], &EventData.Cost,
        sizeof(EventData.Cost), 0);

        MyEvent.Data[0].DataPtr = (ULONG64) &(EventData.Cost);
        MyEvent.Data[0].Length = sizeof(EventData.Cost);
        MyEvent.Data[1].DataPtr = (ULONG64) &(EventData.Indices);
        MyEvent.Data[1].Length = sizeof(EventData.Indices);
        MyEvent.Data[2].DataPtr = (ULONG64) &(EventData.Signature);
        MyEvent.Data[2].Length = (ULONG)(wcslen(EventData.Signature) + 1) *
sizeof(WCHAR);
        MyEvent.Data[3].DataPtr = (ULONG64) &(EventData.IsComplete);
        MyEvent.Data[3].Length = sizeof(EventData.IsComplete);
        MyEvent.Data[4].DataPtr = (ULONG64) &(EventData.ID);
        MyEvent.Data[4].Length = sizeof(EventData.ID);
        MyEvent.Data[5].DataPtr = (ULONG64) &(EventData.Size);
        MyEvent.Data[5].Length = sizeof(EventData.Size);

        // Write the event.

        status = TraceEvent(g_SessionHandle, &(MyEvent.Header));

        if (ERROR_SUCCESS != status)
        {
            // Decide how you want to handle failures. Typically, you do not
            // want to terminate the application because you failed to
            // log an event. If the error is a memory failure, you may
            // may want to log a message to the system event log or turn
            // off logging.

            wprintf(L"TraceEvent() event failed with %lu\n", status);
            g_TraceOn = FALSE;
        }
    }

cleanup:

    if (RegistrationHandle)
    {
        UnregisterTraceGuids(RegistrationHandle);
    }
}

// The callback function that receives enable/disable notifications
// from one or more ETW sessions. Because more than one session
// can enable the provider, this example ignores requests from other
// sessions if it is already enabled.

ULONG WINAPI ControlCallback(
    WMIDPREQUESTCODE RequestCode,
    PVOID Context,
    ULONG* Reserved,
    PVOID Header
)
{

```

```

UNREFERENCED_PARAMETER(Context);
UNREFERENCED_PARAMETER(Reserved);

ULONG status = ERROR_SUCCESS;
TRACEHANDLE TempSessionHandle = 0;

switch (RequestCode)
{
    case WMI_ENABLE_EVENTS: // Enable the provider.
    {
        SetLastError(0);

        // If the provider is already enabled to a provider, ignore
        // the request. Get the session handle of the enabling session.
        // You need the session handle to call the TraceEvent function.
        // The session could be enabling the provider or it could be
        // updating the level and enable flags.

        TempSessionHandle = GetTraceLoggerHandle(Header);
        if (INVALID_HANDLE_VALUE == (HANDLE)TempSessionHandle)
        {
            wprintf(L"GetTraceLoggerHandle failed. Error code is
%lu.\n", status = GetLastError());
            break;
        }

        if (0 == g_SessionHandle)
        {
            g_SessionHandle = TempSessionHandle;
        }
        else if (g_SessionHandle != TempSessionHandle)
        {
            break;
        }

        // Get the severity level of the events that the
        // session wants you to log.

        g_EnableLevel = GetTraceEnableLevel(g_SessionHandle);
        if (0 == g_EnableLevel)
        {
            // If zero, determine whether the session passed zero
            // or an error occurred.

            if (ERROR_SUCCESS == (status = GetLastError()))
            {
                // Decide what a zero enable level means to your
provider.
                // For this example, it means log all events.
                ;
            }
            else
            {
                wprintf(L"GetTraceEnableLevel failed with, %lu.\n",
status);
            }
        }
    }
}

```

```

        break;
    }

    // Get the enable flags that indicate the events that the
    // session wants you to log. The provider determines the
    // flags values. How it articulates the flag values and
    // meanings to perspective sessions is up to it.

    g_EnableFlags = GetTraceEnableFlags(g_SessionHandle);
    if (0 == g_EnableFlags)
    {
        // If zero, determine whether the session passed zero
        // or an error occurred.

        if (ERROR_SUCCESS == (status = GetLastError()))
        {
            // Decide what a zero enable flags value means to your
provider.
            ;
        }
        else
        {
            wprintf(L"GetTraceEnableFlags failed with, %lu.\n",
status);
            break;
        }
    }

    g_TraceOn = TRUE;
    break;
}

case WMI_DISABLE_EVENTS: // Disable the provider.
{
    // Disable the provider only if the request is coming from the
    // session that enabled the provider.

    TempSessionHandle = GetTraceLoggerHandle(Header);
    if (INVALID_HANDLE_VALUE == (HANDLE)TempSessionHandle)
    {
        wprintf(L"GetTraceLoggerHandle failed. Error code is
%lu.\n", status = GetLastError());
        break;
    }

    if (g_SessionHandle == TempSessionHandle)
    {
        g_TraceOn = FALSE;
        g_SessionHandle = 0;
    }
    break;
}

default:

```

```
    {
        status = ERROR_INVALID_PARAMETER;
        break;
    }

    return status;
}
```

# Writing WPP Events

Article • 01/07/2021 • 2 minutes to read

The Windows software trace processor (WPP) provides an efficient mechanism to log events that occur during the execution of an application or driver. WPP uses the ETW API.

For details on using WPP to log events, see [WPP Software Tracing](#) and [Software Tracing FAQ](#) in the Microsoft Windows Driver Development Kit (DDK).

# Obtaining a Provider and Class GUID

Article • 01/07/2021 • 2 minutes to read

To obtain a provider GUID or event trace class GUIDs, you can use the Uuidgen.exe or Guidgen.exe tool.

If you use the Uuidgen.exe tool, use the -d option to create a DEFINE\_GUID C macro, as shown in the following example. For information about using the Uuidgen.exe tool, use the -? option. If you use the DEFINE\_GUID C macro to define your GUID, you must include #define INITGUID before your GUID definitions, as shown in the following example.

syntax

```
#define INITGUID

DEFINE_GUID (
    ProviderGuid,
    0xf4dc272d,
    0x88dd,
    0x4472,
    0xa3, 0xb1, 0xcb, 0x6d, 0xa4, 0x86, 0xf0, 0xbe
);
```

The Microsoft Windows Software Development Kit (SDK) and Microsoft Visual Studio include the Guidgen.exe tool. The Guidgen.exe tool has a user interface that lets you select from several formats. You should use the format that creates a static constant GUID, as shown in the following example.

syntax

```
// {7C214FB1-9CAC-4b8d-BAED-7BF48BF63BB3}
static const GUID ProviderGuid =
{ 0x7c214fb1, 0x9cac, 0x4b8d, { 0xba, 0xed, 0x7b, 0xf4, 0x8b, 0xf6, 0x3b,
0xb3 } };
```

# Provider Traits

Article • 01/07/2021 • 2 minutes to read

Provider Traits are a method of attaching more data to an individual provider registration. They can be used for manifest-based or TraceLogging providers. This currently includes support for adding a Provider Name and/or Provider Group to an individual provider registration. More trait types are likely to be added in the future. This information is stored in the kernel as a binary blob of a set format.

Traits can only be set once for a registration. Any further attempts to set the traits on that registration will fail.

To set Provider Traits on a manifest-based provider, call the [EventSetInformation](#) function with the EventProviderSetTraits information class. The EventInformation buffer should contain a binary blob of the following format:

syntax

```
{  
    UINT16 TraitsSize    // Total size of the traits including this field  
    CHAR[] ProviderName // Null terminated utf-8 provider name  
    TRAIT[] Traits      // Zero or more individual traits  
}
```

Individual traits should be in the following format:

syntax

```
TRAIT {  
    UINT16 TraitSize // Size of this individual trait including this field  
    UINT8 Type       // ETW_PROVIDER_TRAIT_TYPE  
    BYTE[] Data  
}
```

From the individual trait, ETW\_PROVIDER\_TRAIT\_TYPE is defined as:

syntax

```
typedef enum {  
    EtwProviderTraitTypeGroup = 1,  
    EtwProviderTraitTypeMax  
} ETW_PROVIDER_TRAIT_TYPE;
```

TraceLogging providers automatically set the Provider Traits when the [TraceLoggingRegister](#) function is called. The TraceLogging provider's name will always

be included in its traits. A group can be set on a TraceLogging provider using the [TraceLoggingOptionGroup](#) macro in the provider definition.

## Custom Traits

Although most of the 255 possible trait types are not yet defined, trait types 1-127 are reserved for definition by Microsoft. The remaining higher indexed type values can be used by external developers as they see fit. Anyone considering adding their own custom traits to their provider should try to keep their total trait size under 256 bytes for the following reasons:

- The traits are included in every event written for the provider. Large traits could lead to very large log files.
- The traits are stored in nonpaged kernel pool for the lifetime of the provider.

## Provider Groups

A provider group is a GUID-defined controllable entity much like a provider itself. The key difference is that while a provider GUID is used to control registrations of just its provider, a group will control all of its member registrations. For example, enabling a provider group with a given keyword and level will enable all of the groups member registrations with that keyword and level.

Group membership may be restricted by permissions. If the caller of [EventSetInformation](#) doesn't have permissions to join the specified group, then membership will be denied.

In some cases the trace session controller may want to exclude a few providers from its enable of a group. This can be done by setting a disallow list. A disallow list is a list of provider GUIDs that will not be enabled based on the group settings for a single logging session. Disallow lists can be changed dynamically with [TraceSetInformation](#) and the [TraceSetDisallowList](#) information class.

While most enable actions can be done for Provider Groups in a similar manner to individual providers, there are some exceptions. Exceptions include:

- Provider Groups cannot be controlled by private trace sessions.
- Event Name, Event ID, and Payload filters are not applicable to Provider Groups as they assume specific information of an individual provider.



# Writing Related Events in an End-to-End Scenario

Article • 01/07/2021 • 2 minutes to read

ETW provides a way to group related events from more than one component. For example, if several components (either on the same computer or on different computers) are involved in processing the same data, and each component logs events for their portion of the activity, you can group all of the related events together. This will allow a consumer to consume all of the events, from the beginning of the process to the end of the process.

To write related events in a [manifest-based provider](#), see [Writing Related Events in a Manifest-based Provider](#).

To write related events in a [classic provider](#), see [Writing Related Events in a Classic Provider](#).

# Writing Related Events in a Manifest-based Provider

Article • 01/07/2021 • 2 minutes to read

Use the [EventWriteTransfer](#) function when several components want to relate their events in an end-to-end tracing scenario. For example, components A, B and C perform work on a related activity and want to link all the events related to that activity.

ETW uses thread local storage to make the previous component's activity identifier available to the next component. The component retrieves the previous component's identifier from the local storage and sets the related activity identifier to it. The consumer can then use the related activity identifier to walk the chain of the events from one component to the next.

# Writing Related Events in a Classic Provider

Article • 01/07/2021 • 2 minutes to read

Classic providers use the [TraceEventInstance](#) function to trace events that are part of a single transaction. You can also use this function to trace parent/child events.

Before calling the [TraceEventInstance](#) function, you must first call the [CreateTraceInstanceId](#) function to obtain a transaction identifier. This function generates a unique transaction identifier, and maps it to a registered class GUID handle. The handles for registered class GUIDs are available in the [RegHandle](#) members of [TRACE\\_GUID\\_REGISTRATION](#) structures, after calling the [RegisterTraceGuids](#) function. The transaction identifier is placed in the [Instanceld](#) member of an [EVENT\\_INSTANCE\\_INFO](#) structure that you pass to the [CreateTraceInstanceId](#) function.

The [EVENT\\_INSTANCE\\_HEADER](#) structure that is passed to the [TraceEventInstance](#) function is similar to the [EVENT\\_TRACE\\_HEADER](#) structure (see [Tracing Events](#)), except that it contains additional information relating to instances, and does not contain a [Guid](#) member.

Event instances can be used to establish a hierarchical relationship between events. The [TraceEventInstance](#) function accepts instance-specific information from two event instances. The *pInstInfo* parameter points to the [EVENT\\_INSTANCE\\_INFO](#) structure of the event instance, and the *pParentInstInfo* parameter points to the [EVENT\\_INSTANCE\\_INFO](#) structure of a parent event instance. The definition of a "parent" event instance is application-defined; the parent can be any instance that has already been generated.

# Publishing Your Event Schema

Article • 01/07/2021 • 2 minutes to read

For information on publishing your event schema for a [manifest-based](#) provider, see [Publishing Your Event Schema for a Manifest-based Provider](#).

For information on publishing your event schema for a [classic](#) provider, see [Publishing Your Event Schema for a Classic Provider](#).

# Publishing Your Event Schema for a Manifest-based Provider

Article • 01/07/2021 • 2 minutes to read

[Manifest-based](#) providers use a manifest to publish the schema for their events. The manifest is embedded in the provider binary, which means that the provider must be available on the computer for the consumer to consume its events. For complete details on writing a manifest, see [Writing an Instrumentation Manifest](#).

The following manifest defines the events that are used in examples in the [Providing Events](#) and [Consuming Events](#) section of the document.

## syntax

```
<!-- <?xml version="1.0" encoding="UTF-16"?> -->
<instrumentationManifest
    xmlns="http://schemas.microsoft.com/win/2004/08/events"
    xmlns:win="https://manifests.microsoft.com/win/2004/08/windows/events"
    xmlns:xs="https://www.w3.org/2001/XMLSchema"
    >

    <instrumentation>
        <events>

            <provider name="Microsoft-Windows-ETWProvider"
                guid="{D8909C24-5BE9-4502-98CA-AB7BDC24899D}"
                symbol="ProviderGuid"

                resourceFileName="c:\code\etw\v2provider\debug\v2provider.exe"

                messageFileName="c:\code\etw\v2provider\debug\v2provider.exe"
                message="$(string.Provider.Name)"
                >

            <keywords>
                <keyword name="Read" symbol="READ_KEYWORD" mask="0x1" />
                <keyword name="Write" symbol="WRITE_KEYWORD" mask="0x2" />
                <keyword name="Local" symbol="LOCAL_KEYWORD" mask="0x4" />
                <keyword name="Remote" symbol="REMOTE_KEYWORD" mask="0x8" />
            </keywords>

            <maps>
                <valueMap name="TransferType">
                    <map value="1" message="$(string.Map.Download)"/>
                    <map value="2" message="$(string.Map.Upload)"/>
                    <map value="3" message="$(string.Map.UploadReply)"/>
                </valueMap>
            </maps>
        </events>
    </instrumentation>
</instrumentationManifest>
```

```

        </valueMap>

        <bitMap name="DaysOfTheWeek">
            <map value="0x1" message="$string.Map.Sunday"/>
            <map value="0x2" message="$string.Map.Monday"/>
            <map value="0x4" message="$string.Map.Tuesday"/>
            <map value="0x8" message="$string.Map.Wednesday"/>
            <map value="0x10" message="$string.Map.Thursday"/>
            <map value="0x20" message="$string.Map.Friday"/>
            <map value="0x40" message="$string.Map.Saturday"/>
        </bitMap>
    </maps>

    <templates>

        <template tid="TransferTemplate">
            <data name="Image" inType="win:Pointer"/>
            <data name="Scores" inType="win:UInt16" count="3"/>
            <data name="ID" inType="win:GUID"/>
            <data name="Certificate" inType="win:Binary"
length="11" />
            <data name="IsLocal" inType="win:Boolean" />
            <data name="Path" inType="win:UnicodeString" />

            <data name="ValuesCount" inType="win:UInt16" />
            <struct name="Values" count="ValuesCount" >
                <data name="Name" inType="win:UnicodeString" />
                <data name="Value" inType="win:UInt16" />
            </struct>

            <data name="Day" inType="win:UInt32"
map="DaysOfTheWeek"/>
            <data name="Transfer" inType="win:UInt32"
map="TransferType"/>

            <UserData>
                <EventData xmlns="ProviderNamespace">
                    <Transfer> %10 </Transfer>
                    <Day> %9 </Day>
                    <ValuesCount> %7 </ValuesCount>
                    <Values> %8 </Values>
                    <Path> %6 </Path>
                    <IsLocal> %5 </IsLocal>
                    <Scores> %2 </Scores>
                    <Image> %1 </Image>
                    <Certificate> %4 </Certificate>
                    <ID> %3 </ID>
                </EventData>
            </UserData>
        </template>

    </templates>

    <events>
        <event value="1"

```

```
        level="win:Informational"
        template="TransferTemplate"
        symbol="TransferEvent"
        message ="$(string.Event.WhenToTransfer)"
        keywords="Read Local" />
    </events>

</provider>

</events>

</instrumentation>

<localization>
    <resources culture="en-US">
        <stringTable>

            <string id="Provider.Name" value="Microsoft-Windows-
ETWProvider"/>

            <string id="Map.Download" value="Download"/>
            <string id="Map.Upload" value="Upload"/>
            <string id="Map.UploadReply" value="Upload-reply"/>

            <string id="Map.Sunday" value="Sunday"/>
            <string id="Map.Monday" value="Monday"/>
            <string id="Map.Tuesday" value="Tuesday"/>
            <string id="Map.Wednesday" value="Wednesday"/>
            <string id="Map.Thursday" value="Thursday"/>
            <string id="Map.Friday" value="Friday"/>
            <string id="Map.Saturday" value="Saturday"/>

            <string id="Event.WhenToTransfer" value="The %10 transfer
will occur %9."/>

        </stringTable>
    </resources>
</localization>

</instrumentationManifest>
```

# Publishing Your Event Schema for a Classic Provider

Article • 01/07/2021 • 6 minutes to read

[Classic](#) providers should use [Managed Object Format](#) (MOF) to publish the layout of their event data. Consumers can then read the published layout from WMI at runtime and use it to read the event data.

If you use MOF to publish the layout of your event data in WMI, you typically create the following three types of MOF classes in the root\wmi namespace:

- A provider MOF class
- One or more event MOF classes
- One or more event type MOF classes

## Provider MOF classes

If you publish the layout of your event data, you must create a MOF class that identifies your provider. This class must derive from the [EventTrace](#) MOF class and must be empty (no properties or methods). The class must also include the [Guid](#) qualifier which uniquely identifies the provider. This is the same GUID you use when you calling the [RegisterTraceGuids](#) function to register your provider.

## Event MOF classes

An event MOF class defines a class of events that the provider provides. This class derives from the provider MOF class and must be empty (no properties or methods). The class must also include the [Guid](#) qualifier which uniquely identifies the class of events that its child classes define. The provider uses this same GUID when setting the [Guid](#) member of the [EVENT\\_TRACE\\_HEADER](#) structure.

## Event type MOF classes

An event type MOF class defines the actual event data. This class derives from its parent event MOF class. When naming the event type MOF class, the convention is to use the event MOF class name at the beginning of the event type MOF class name. For example, if the event MOF class name is HWConfig and the event type MOF class represents CPU information, you should name the event type MOF class, HWConfig\_CPU.

Use the **EventType** qualifier on the event type MOF class to identify the event type. If multiple event types use the same event data, they can use the same MOF class. The provider uses the same event type value to identify the event when setting the **Class.Type** member of the **EVENT\_TRACE\_HEADER** structure.

The event type MOF class contains properties. The order of these properties define the layout of the event data. The following table identifies the data types and qualifiers you can use to define the properties. For more information on the property and class qualifiers that you can use, see [Event Tracing MOF Qualifiers](#).

| <b>Data type</b>  | <b>Qualifiers</b>            | <b>Description</b>  |
|-------------------|------------------------------|---|
| sint8,<br>uint8   | Format                       | Declares a 1-byte decimal integer. To declare an ANSI character, use the <b>Format</b> qualifier and set its value to "c".  |
| sint16,<br>uint16 | Format                       | Declares a 2-byte decimal integer. To indicate the number is a hexadecimal number, use the <b>Format</b> qualifier. For example, format("x").   |
| sint32,<br>uint32 | Format                       | Declares a 4-byte decimal integer. To indicate the number is a hexadecimal number, use the <b>Format</b> qualifier and set its value to "x".  |
| sint64,<br>uint64 | Format                       | Declares a 8-byte decimal integer. To indicate the number is a hexadecimal number, use the <b>Format</b> qualifier and set its value to "x".  |
| boolean           |                              | Declares a Boolean value. The event consumer should interpret the value as BOOL (4-byte integer).   |
| char16            |                              | Declares a wide character. The event consumer should interpret char16 arrays in kernel events as wide character strings. (Use the array size to copy the string. Some strings may contain leading NULL characters.)   |
| object            | Extension                    | Declares a binary blob. The <b>Extension</b> qualifier indicates the type of data in the blob.  |
| string            | Format,<br>StringTermination | Declares a string value. To indicate the string is a wide-character string use the <b>Format</b> qualifier and set its value to "w". The string is considered an ANSI string if you do not specify the <b>Format</b> qualifier. To indicate how the string is terminated, use the <b>StringTermination</b> qualifier. |

To specify an array, you can use square brackets, []. The square brackets can include the size of the array. For example:

```
[WmiDataId(1), read] uint8 MyGuid[16];
```

You can also use the **Max** qualifier to specify the size of an array. For example:

```
[WmiDataId(1), Max(16), read] uint8 MyGuid[];
```

If you include the size of the array in the square brackets, the MOF compiler generates the **Max** qualifier for you.

It is important that you use the **Description** qualifier for each property. The description should contain a display name that the consumer can use when displaying the property values.

The following example shows the contents of a MOF file that describes a provider, event, and event type MOF class.

syntax

```
#pragma namespace("\\\\.\\\\root\\\\wmi")

[dynamic: ToInstance, Description("Defines my event provider"),
 Guid("{7C214FB1-9CAC-4b8d-BAED-7BF48BF63BB3}")]
class MyProvider : EventTrace
{
};

[dynamic: ToInstance, Description("Defines a category of events that my
provider logs."): Amended,
 Guid("{B49D5931-AD85-4070-B1B1-3F81F1532875}")]
class MyCategory : MyProvider
{
};

[dynamic: ToInstance, Description("Defines an event within the category of
events that my provider logs."): Amended,
 EventType(1)]
class MyCategory_MyEvent : MyCategory
{
    [WmiDataId(1), Description("Cost factor"): Amended, read] sint32 Cost;
    [WmiDataId(2), Description("Index values"): Amended, read] uint32
Indices[3];
    [WmiDataId(3), Description("Signature"): Amended, read,
StringTermination("NullTerminated"), Format("w")] string Signature;
    [WmiDataId(4), Description("Is complete copy"): Amended, read] boolean
IsComplete;
    [WmiDataId(5), Description("Class identifier"): Amended, read,
Extension("Guid")] object ID;
};
```

Note that the provider, event, and event type MOF class names must be unique within the entire namespace. To avoid naming conflicts, you should use unique and descriptive name for all class names. Class properties should also be descriptive and unique within its class hierarchy—a child class that contains the same property name as a parent class overwrites the property of the parent class.

After defining your MOF classes, use the MOF compiler to generate your event schema and add it to the CIM repository. Consumers can then read the schema from the repository and programmatically read the event data. For a complete description of the MOF syntax and using the MOF compiler (Mofcomp.exe) to add your MOF classes to the CIM repository, see [Managed Object Format](#). For information on using Wbemtest.exe to access the CIM repository, see [Windows Management Instrumentation \(WMI\)](#).

## Versioning MOF class

If you add or change an event type MOF class, the convention is to version both the event MOF class and its child event type MOF classes. To version the current event MOF class, append \_Vn to the class name, where n is an incremental number starting at 0. If this is the first revision to the class, append \_V0 to the class name. You must also add the **EventVersion** qualifier to the class. Use the same version number you used in the class name for the value of the **EventVersion** qualifier.

The new version of the event MOF class must use the same name and **Guid** qualifier as the original class. The new class may optionally add the **EventVersion** qualifier. The event MOF class that does not contain the **EventVersion** qualifier is considered the latest version, or if all the versions of the class contain an **EventVersion** qualifier, then the class with the highest version number is considered the latest version. The provider uses the **Class.Version** member of the [EVENT\\_TRACE\\_HEADER](#) structure to identify the version of the event included in the trace.

The following example shows how to version an event MOF class.

### Syntax

```
#pragma namespace("\\\\.\\root\\\\wmi")
#pragma classflags("forceupdate")

[dynamic: ToInstance, Description("Defines my event provider"),
 Guid("{7C214FB1-9CAC-4b8d-BAED-7BF48BF63BB3}")]
class MyProvider : EventTrace
{
};

[dynamic: ToInstance, Description("Defines a category of events that my
provider logs."),
```

```

        Guid("{B49D5931-AD85-4070-B1B1-3F81F1532875}"),
        EventVersion(1)]
class MyCategory : MyProvider
{
};

[dynamic: ToInstance, Description("Defines an event within the category of
events that my provider logs."): Amended,
 EventType(1),
 EventName("MyEvent")]
class MyCategory_MyEvent : MyCategory
{
    [WmiDataId(1), Description("Cost factor"): Amended, read] sint32 Cost;
    [WmiDataId(2), Description("Index values"): Amended, read] uint32
Indices[3];
    [WmiDataId(3), Description("Signature"): Amended, read,
StringTermination("NullTerminated"), Format("w")] string Signature;
    [WmiDataId(4), Description("Is complete copy"): Amended, read] boolean
IsComplete;
    [WmiDataId(5), Description("Identifier"): Amended, read,
Extension("Guid")] object ID;
    [WmiDataId(6), Description("Buffer Size"): Amended, read] uint32 Size;
};

[dynamic: ToInstance, Description("Defines a category of events that my
provider logs."): Amended,
 Guid("{B49D5931-AD85-4070-B1B1-3F81F1532875}"),
 EventVersion(0)]
class MyCategory_V0 : MyProvider
{
};

[dynamic: ToInstance, Description("Defines an event within the category of
events that my provider logs."): Amended,
 EventType(1)]
class MyCategory_V0_MyEvent : MyCategory_V0
{
    [WmiDataId(1), Description("Cost factor"): Amended, read] sint32 Cost;
    [WmiDataId(2), Description("Index values"): Amended, read] uint32
Indices[3];
    [WmiDataId(3), Description("Signature"): Amended, read,
StringTermination("NullTerminated"), Format("w")] string Signature;
    [WmiDataId(4), Description("Is complete copy"): Amended, read] boolean
IsComplete;
    [WmiDataId(5), Description("Identifier"): Amended, read,
Extension("Guid")] object ID;
};

```

# Consuming Events (Event Tracing)

Article • 01/07/2021 • 2 minutes to read

Event trace consumers can process events from one or more providers. Consumers can process events from a log file or in real time. You can consume events in real time only if the controller specifies the real time logging mode for the session. For performance reasons, real-time processing is not recommended prior to Windows Vista.

To specify the trace session from which you want to process events, you use the [EVENT\\_TRACE\\_LOGFILE](#) structure. You must initialize a copy of this structure for each log file or real time session that you want to process.

To consume events from a log file, set the **LogFileName** member to the name of the log file. To consume events from real time session, set the **LoggerName** member to the session name. You also use this structure to specify the [BufferCallback](#) callback and the [EventCallback](#) or [EventRecordCallback](#) callback used to process the events.

- [EventRecordCallback](#)—Receives and processes all events (including the header event) from one or more log files and a real-time session. You implement this callback if you use the trace data helper functions to parse the event data or you want to retrieve metadata about the event.
- [EventCallback](#)—Receives and processes all events (including the header event) from one or more log files and a real-time session.
- [BufferCallback](#)—Receives and processes summary information about the current buffer, such as events lost. ETW calls the callback after delivering all events in the buffer to the consumer. The consumer can also use this callback to cancel event processing; however, if you are consuming events in real time, ETW delivers events until the controller stops the session.

After defining one or more trace sessions, call the [OpenTrace](#) function for each trace session that you want to process; you can process events from one or more log files, but from only one real-time session. You then pass the list of trace session handles that [OpenTrace](#) returns to the [ProcessTrace](#) function. The [ProcessTrace](#) function combines the events, sorts them into chronological order, and then delivers them to the callbacks one at a time. The events can be filtered to include only those that fall into a specific time frame using the *StartTime* and *EndTime* parameters. The [ProcessTrace](#) function blocks the thread until your consumer processes all events in the trace sessions, the [BufferCallback](#) returns **FALSE**, or you call [CloseTrace](#).

**Prior to Windows Vista:** You can call [CloseTrace](#) only after [ProcessTrace](#) returns.

For an example that shows how to consume events published using a manifest, MOF, or TMF files, see [Retrieving Event Data Using TDH](#). Note that beginning with Windows Vista, you should use the trace data helper (TDH) functions to consume events.

For an example that shows how to consume events published using MOF, see [Retrieving Event Data Using MOF](#).

# Retrieving Event Data

Article • 06/18/2021 • 2 minutes to read

To consume event specific data, the consumer must know the format of the event data. This is not an issue if you are consuming events from your own provider because you know the format of the data. However, to consume an event from any provider, the provider must publish the layout of the event data. For an overview of how event metadata operates as well as how to publish your event metadata so others can use it, see [Event Metadata Overview](#).

To consume events on Windows Vista that were published using a manifest, MOF, or TMF files, as well as TraceLogging events, see [Retrieving Event Data Using TDH](#).

To consume events prior to Windows Vista that were published using MOF, see [Retrieving Event Data Using MOF](#).

# Event Metadata Overview

Article • 01/07/2021 • 5 minutes to read

This is an overview of what to expect for the end-to-end journey of an event for each of the four tracing technologies furnished by Microsoft: [TraceLogging](#), [Manifest-based](#), [WPP](#), and [MOF](#). It approaches the problem with respect to both an individual event's payload and the accompanying metadata that describes its structure, making it possible to later decode the event into sensible pieces of data. Understanding the journey of each piece of an event is important when choosing which tracing technology is appropriate for a project. There is no one-size-fits-all solution to tracing.

## Event Payloads

For any of the Microsoft-provided tracing technologies, when calling the Write command (for instance [EventWrite](#) for manifest-based or [TraceLoggingWrite](#) for TraceLogging), the data provided for the event at runtime is folded into a contiguous binary blob called the event payload. This is separate from the event schema or event metadata (discussed below), which describes the layout of this binary blob for decoding tools. How exactly the generation of the event payload happens varies depending on the tracing technology used and ultimately whether the event is classic ([TraceEvent](#) style) or modern ([EventWrite](#) style).

For classic events, the flag in [EVENT\\_TRACE\\_HEADER](#) is passed into [TraceEvent](#) which determines how this occurs:

- If [WNODE\\_FLAG\\_USE\\_MOF\\_PTR](#) is specified, an array of [MOF\\_FIELDS](#) follows the [EVENT\\_TRACE\\_HEADER](#) in memory. The ETW runtime will concatenate the event data as specified in these fields to form the event payload.
- If [WNODE\\_FLAG\\_USE\\_MOF\\_PTR](#) is not specified, the event payload is constructed by the user directly in memory following the [EVENT\\_TRACE\\_HEADER](#).

Both MOF and WPP are classic providers. However, the WPP implementation takes care of all of this for you.

For non-classic events, a number of [EVENT\\_DATA\\_DESCRIPTOR](#)s are passed into [EventWrite](#). The ETW runtime will concatenate the event data as specified in these descriptors to form the event payload.

Both Manifest-based and TraceLogging technologies are generally modern providers. With manifest-based, if you let mc.exe generate logging code for you (um or km switch),

payload generation is taken care of for you. Payload generation is always taken care of by TraceLogging.

The end result is that a payload for an event is generated at runtime. All payloads are fundamentally similar in that they are binary blobs of data containing only the field data for that particular event, regardless of the tracing technology used and which field types are supported by that tracing technology. Without the event metadata, the event payload is meaningless and unintelligible.

The ETW runtime's duty is then to carry this event blob from the provider to the consumer, where it is re-associated with its metadata and becomes decodable. The ETW runtime will add a bit of extra metadata indicating the circumstances of the payload -- for instance things like the timestamp, thread ID, and keywords of the event. This information is relevant to how the event was routed through the runtime, and it is also useful for understanding the identity and context of the event at decoding time. It is eventually surfaced as part of the [EVENT\\_TRACE](#) or [EVENT\\_RECORD](#) for the consumer

## Event Metadata

The journey for event metadata is different depending on which tracing technology is used, and it is one of the largest considerations when comparing each technology.

### MOF

Event metadata is authored by hand by the creator of the event in the form of a special MOF schema in a .mof file. The schema authored must match the payload that is logged in order for the event to be correctly decoded by consumers. Event decoders require that the .mof file is registered on the system using [mofcomp.exe](#). For more information on publishing MOF schemas, see [Publishing Your Event Schema for a Classic Provider](#).

### WPP

Event metadata is automatically generated and stored in your binary's .pdb by tracewpp.exe during the build process. This metadata can later be extracted in the form of a standalone .tmf file by way of tracepdb.exe. Event decoders typically require either the .pdb or the .tmf file. For more information on tracepdb.exe, see [Tracepdb Overview](#), and for more information on tracewpp.exe, see [TraceWPP task](#).

### Manifest-based

Event definitions are authored in a .man file. Event manifests are run through the manifest compiler ([mc.exe](#)), generating the headers needed for source compilation as well as several .bin binary resource files. These resource files must then be compiled as resources into a binary and the resulting binary placed on the system that intends to decode events from that manifest-based provider. Additionally, the manifest must be installed on the system using [wevtutil.exe](#). Most importantly, the **resourceFileName** and **messageFileName** attributes on the provider XML element in the installed event manifest must correctly identify the full absolute path to the binary in which the resource files were placed. Note that these paths may be overwritten at manifest install time by using the wevtutil.exe switches /rf and /mf. A system that has not correctly and fully performed these steps will be unable to decode events from this provider. Event decoders thus require an installed manifest, and the binary with the associated resources installed in the location specified by resource and message file paths. For more information on publishing manifest-based schemas, see [Publishing Your Event Schema for a Manifest-based Provider](#).

## TraceLogging

All TraceLogging events are self-describing. The event metadata necessary to decode the event is automatically generated and transmitted along with the payload in a compact binary format. Event decoders need only the TraceLogging event and an understanding of the TraceLogging metadata format.

## Configuring TDH for Decoding

There are many decoding tools out there. However, it is recommended that you use TDH where possible, as it decodes all tracing technologies with a unified API. Depending on the type of tracing you are interested in decoding, TDH may require explicit configuration.

## MOF

TDH utilizes MOF decoding data registered on the system using mofcomp.exe. For more information, see [Publishing Your Event Schema for a Classic Provider](#).

## WPP

TDH must be pointed towards either the .pdb file or the associated .tmf for the WPP provider you are trying to decode. This can be performed by calling

[TdhSetDecodingParameter](#). Otherwise, the decoding engine will fall back on the environment variable TRACE\_FORMAT\_SEARCH\_PATH.

## Manifest-based

TDH utilizes all manifest-based providers registered on the system using wevtutil.exe.

## TraceLogging

The newest versions of TDH automatically detect and decode TraceLogging events with no additional steps.

# Retrieving Event Data Using TDH

Article • 08/08/2022 • 2 minutes to read

To consume event specific data, the consumer must know the format of the event data. If the provider used TraceLogging, a manifest, MOF, or TMF files to publish the format of the event data, you can use the trace data helper (TDH) functions to parse the event data. The TDH functions are available beginning with Windows Vista.

The following topics show you how to use TDH to retrieve event data and metadata.

- [Using TdhFormatProperty to Consume Event Data](#)
- [Using TdhGetProperty to Consume Event Data](#)
- [Retrieving Event Metadata](#)
- [Enumerating Providers](#)



# Using TdhFormatProperty to Consume Event Data

Article • 08/08/2022 • 13 minutes to read

The following example shows how to consume event data using the [TdhFormatProperty](#) function.

C++

```
/*
Sample for decoding ETW events using TdhFormatProperty.

This sample demonstrates the following:

- How to process events from ETL files using OpenTrace and ProcessTrace.
- How to get TRACE_EVENT_INFO data for an event using
TdhGetEventInformation.
- How to format a WPP message using TdhGetProperty.
- How to extract property values from non-WPP events.
- How to format property values from non-WPP events using TdhFormatProperty.
*/

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN 1 // Exclude rarely-used APIs from <windows.h>
#endif

#include <windows.h>

#define INITGUID // Ensure that EventTraceGuid is defined.
#include <evntrace.h>
#undef INITGUID

#include <tdh.h>
#include <vector>

#include <wchar.h> // wprintf

#pragma comment(lib, "tdh.lib") // Link against TDH.dll

// Support building this sample using older versions of the Windows SDK:
#define EventNameOffset          ActivityIDNameOffset
#define EventAttributesOffset    RelatedActivityIDNameOffset

/*
Decodes event data using TdhGetEventInformation and TdhFormatProperty.
Prints
the event information to stdout.

We use a context object so we can reuse buffers instead of allocating new
buffers and freeing them for each event.
```

```

*/
class DecoderContext
{
public:

    /*
    Initialize the decoder context.
    Sets up the TDH_CONTEXT array that will be used for decoding.
    */
    explicit DecoderContext(
        _In_opt_ LPCWSTR szTmfSearchPath)
    {
        TDH_CONTEXT* p = m_tdhContext;

        if (szTmfSearchPath != nullptr)
        {
            p->ParameterValue = reinterpret_cast<UINT_PTR>(szTmfSearchPath);
            p->ParameterType = TDH_CONTEXT_WPP_TMFSEARCHPATH;
            p->ParameterSize = 0;
            p += 1;
        }

        m_tdhContextCount = static_cast<BYTE>(p - m_tdhContext);
    }

    /*
    Decode and print the data for an event.
    Might throw an exception for out-of-memory conditions. Caller should
    catch
        the exception before returning from the ProcessTrace callback.
    */
    void PrintEventRecord(
        _In_ EVENT_RECORD* pEventRecord)
    {
        if (pEventRecord->EventHeader.EventDescriptor.Opcode ==
EVENT_TRACE_TYPE_INFO &&
            pEventRecord->EventHeader.ProviderId == EventTraceGuid)
        {
            /*
            The first event in every ETL file contains the data from the
            file header.
            This is the same data as was returned in the
            EVENT_TRACE_LOGFILEW by
                OpenTrace. Since we've already seen this information, we'll skip
            this
                event.
            */
            return;
        }

        // Reset state to process a new event.
        m_indentLevel = 1;
        m_pEvent = pEventRecord;
        m_pbData = static_cast<BYTE const*>(m_pEvent->UserData);
        m_pbDataEnd = m_pbData + m_pEvent->UserDataLength;
    }
}

```

```

    m_pointerSize =
        m_pEvent->EventHeader.Flags & EVENT_HEADER_FLAG_32_BIT_HEADER
        ? 4
        : m_pEvent->EventHeader.Flags & EVENT_HEADER_FLAG_64_BIT_HEADER
        ? 8
        : sizeof(void*); // Ambiguous, assume size of the decoder's
pointer.

        // There is a lot of information available in the event even without
decoding,
        // including timestamp, PID, TID, provider ID, activity ID, and the
raw data.

        // Show the event timestamp.
        PrintFileTime(reinterpret_cast<FILETIME const&>(m_pEvent-
>EventHeader.TimeStamp));

        if (IsWppEvent())
        {
            PrintWppEvent();
        }
        else
        {
            PrintNonWppEvent();
        }
    }

private:

/*
Print the primary properties for a WPP event.
*/
void PrintWppEvent()
{
    /*
    TDH supports a set of known properties for WPP events:
    - "Version": UINT32 (usually 0)
    - "TraceGuid": GUID
    - "GuidName": UNICODESTRING (module name)
    - "GuidTypeName": UNICODESTRING (source file name and line number)
    - "ThreadId": UINT32
    - "SystemTime": SYSTEMTIME
    - "UserTime": UINT32
    - "KernelTime": UINT32
    - "SequenceNum": UINT32
    - "ProcessId": UINT32
    - "CpuNumber": UINT32
    - "Indent": UINT32
    - "FlagsName": UNICODESTRING
    - "LevelName": UNICODESTRING
    - "FunctionName": UNICODESTRING
    - "ComponentName": UNICODESTRING
    - "SubComponentName": UNICODESTRING
    - "FormattedMessage": UNICODESTRING
    - "RawSystemTime": FILETIME
}

```

```

        - "ProviderGuid": GUID (usually 0)
    */

    // Use TdhGetProperty to get the properties we need.
    wprintf(L" ");
    PrintWppStringProperty(L"GuidName"); // Module name (WPP's
"CurrentDir" variable)
    wprintf(L" ");
    PrintWppStringProperty(L"GuidTypeName"); // Source code file name +
line number
    wprintf(L" ");
    PrintWppStringProperty(L"FunctionName");
    wprintf(L"\n");
    PrintIndent();
    PrintWppStringProperty(L"FormattedMessage");
    wprintf(L"\n");
}

/*
Print the value of the given UNICODESTRING property.
*/
void PrintWppStringProperty(_In_z_ LPCWSTR szPropertyName)
{
    PROPERTY_DATA_DESCRIPTOR pdd = { reinterpret_cast<UINT_PTR>
(szPropertyName) };

    ULONG status;
    ULONG cb = 0;
    status = TdhGetPropertySize(
        m_pEvent,
        m_tdhContextCount,
        m_tdhContextCount ? m_tdhContext : nullptr,
        1,
        &pdd,
        &cb);
    if (status == ERROR_SUCCESS)
    {
        if (m_propertyBuffer.size() < cb / 2)
        {
            m_propertyBuffer.resize(cb / 2);
        }

        status = TdhGetProperty(
            m_pEvent,
            m_tdhContextCount,
            m_tdhContextCount ? m_tdhContext : nullptr,
            1,
            &pdd,
            cb,
            reinterpret_cast<BYTE*>(m_propertyBuffer.data()));
    }

    if (status != ERROR_SUCCESS)
    {
        wprintf(L"[TdhGetProperty(%ls) error %u]", szPropertyName,

```

```

status);
    }
    else
    {
        // Print the FormattedString property data (nul-terminated
        // wchar_t string).
        wprintf(L"%ls", m_propertyBuffer.data());
    }
}

/*
Use TdhGetEventInformation to obtain information about this event
(including the names and types of the event's properties). Print some
basic information about the event (provider name, event name), then
print
each property (using TdhFormatProperty to format each property value).
*/
void PrintNonWppEvent()
{
    ULONG status;
    ULONG cb;

    // Try to get event decoding information from TDH.
    cb = static_cast<ULONG>(m_teibuffer.size());
    status = TdhGetEventInformation(
        m_pEvent,
        m_tdhContextCount,
        m_tdhContextCount ? m_tdhContext : nullptr,
        reinterpret_cast<TRACE_EVENT_INFO*>(m_teibuffer.data()),
        &cb);
    if (status == ERROR_INSUFFICIENT_BUFFER)
    {
        m_teibuffer.resize(cb);
        status = TdhGetEventInformation(
            m_pEvent,
            m_tdhContextCount,
            m_tdhContextCount ? m_tdhContext : nullptr,
            reinterpret_cast<TRACE_EVENT_INFO*>(m_teibuffer.data()),
            &cb);
    }

    if (status != ERROR_SUCCESS)
    {
        // TdhGetEventInformation failed so there isn't a lot we can do.
        // The provider ID might be helpful in tracking down the right
        // manifest or TMF path.
        wprintf(L" ");
        PrintGuid(m_pEvent->EventHeader.ProviderId);
        wprintf(L"\n");
    }
    else
    {
        // TDH found decoding information. Print some basic info about
        // the event,
        // then format the event contents.
    }
}

```

```

TRACE_EVENT_INFO const* const pTei =
    reinterpret_cast<TRACE_EVENT_INFO const*>
(m_teiBuffer.data());

    if (pTei->ProviderNameOffset != 0)
    {
        // Event has a provider name -- show it.
        wprintf(L" %ls", TeiString(pTei->ProviderNameOffset));
    }
    else
    {
        // No provider name so print the provider ID.
        wprintf(L" ");
        PrintGuid(m_pEvent->EventHeader.ProviderId);
    }

        // Show core important event properties - try to show some kind
of "event name".
    if (pTei->DecodingSource == DecodingSourceWBEM ||
        pTei->DecodingSource == DecodingSourceWPP)
    {
        // OpcodeName is usually the best "event name" property for
WBEM/WPP events.
        if (pTei->OpcodeNameOffset != 0)
        {
            wprintf(L" %ls", TeiString(pTei->OpcodeNameOffset));
        }

        wprintf(L"\n");
    }
    else
    {
        if (pTei->EventNameOffset != 0)
        {
            // Event has an EventName, so print it.
            wprintf(L" %ls", TeiString(pTei->EventNameOffset));
        }
        else if (pTei->TaskNameOffset != 0)
        {
            // EventName is a recent addition, so not all events
have it.
            // Many events use TaskName as an event identifier, so
print it if present.
            wprintf(L" %ls", TeiString(pTei->TaskNameOffset));
        }

        wprintf(L"\n");

        // Show EventAttributes if available.
        if (pTei->EventAttributesOffset != 0)
        {
            PrintIndent();
            wprintf(L"EventAttributes: %ls\n", TeiString(pTei-
>EventAttributesOffset));
        }
    }
}

```

```

        }

    if (IsStringEvent())
    {
        // The event was written using EventWriteString.
        // We'll handle it later.
    }
    else
    {
        // The event is a MOF, manifest, or TraceLogging event.

        // To help resolve PropertyParamCount and
PropertyParamLength,
        // we will record the values of all integer properties as we
        // reach them. Before we start, clear out any old values and
        // resize the vector with room for the new values.
        m_integerValues.clear();
        m_integerValues.resize(pTei->PropertyCount);

        // Recursively print the event's properties.
        PrintProperties(0, pTei->TopLevelPropertyCount);
    }
}

if (IsStringEvent())
{
    // The event was written using EventWriteString.
    // We can print it whether or not we have decoding information.
    LPCWSTR pchData = static_cast<LPCWSTR>(m_pEvent->UserData);
    unsigned cchData = m_pEvent->UserDataLength / 2;
    PrintIndent();

    // It's probably nul-terminated, but just in case, limit to
cchData chars.
    wprintf(L"%.*ls\n", cchData, pchData);
}
}

/*
Prints out the values of properties from begin..end.
Called by PrintEventRecord for the top-level properties.
If there are structures, this will be called recursively for the child
properties.
*/
void PrintProperties(unsigned propBegin, unsigned propEnd)
{
    TRACE_EVENT_INFO const* const pTei =
        reinterpret_cast<TRACE_EVENT_INFO const*>(m_teibuffer.data());

    for (unsigned propIndex = propBegin; propIndex != propEnd; propIndex
+= 1)
    {
        EVENT_PROPERTY_INFO const& epi = pTei-
>EventPropertyInfoArray[propIndex];

```

```

        // If this property is a scalar integer, remember the value in
case it
        // is needed for a subsequent property's length or count.
if (0 == (epi.Flags & (PropertyStruct | PropertyParamCount)) &&
    epi.count == 1)
{
    switch (epi.nonStructType.InType)
    {
        case TDH_INTYPE_INT8:
        case TDH_INTYPE_UINT8:
            if ((m_pbDataEnd - m_pbData) >= 1)
            {
                m_integerValues[propIndex] = *m_pbData;
            }
            break;
        case TDH_INTYPE_INT16:
        case TDH_INTYPE_UINT16:
            if ((m_pbDataEnd - m_pbData) >= 2)
            {
                m_integerValues[propIndex] =
*reinterpret_cast<UINT16 const UNALIGNED*>(m_pbData);
            }
            break;
        case TDH_INTYPE_INT32:
        case TDH_INTYPE_UINT32:
        case TDH_INTYPE_HEXINT32:
            if ((m_pbDataEnd - m_pbData) >= 4)
            {
                auto val = *reinterpret_cast<UINT32 const
UNALIGNED*>(m_pbData);
                m_integerValues[propIndex] = static_cast<USHORT>(val
> 0xfffffu ? 0xfffffu : val);
            }
            break;
    }
}

PrintIndent();

// Print the property's name.
wprintf(L"%ls:", epi.NameOffset ? TeiString(epi.NameOffset) : L"
(noname)");

m_indentLevel += 1;

// We recorded the values of all previous integer properties
just
// in case we need to determine the property length or count.
USHORT const propLength =
    epi.nonStructType.OutType == TDH_OUTTYPE_IPV6 &&
    epi.nonStructType.InType == TDH_INTYPE_BINARY &&
    epi.length == 0 &&
    (epi.Flags & (PropertyParamLength |
PropertyParamFixedLength)) == 0

```

```

? 16 // special case for incorrectly-defined IPV6 addresses
: (epi.Flags & PropertyParamLength)
? m_integerValues[epi.lengthPropertyIndex] // Look up the
value of a previous property
: epi.length;
USHORT const arrayCount =
(epi.Flags & PropertyParamCount)
? m_integerValues[epi.countPropertyIndex] // Look up the
value of a previous property
: epi.count;

// Note that PropertyParamFixedCount is a new flag and is
ignored
// by many decoders. Without the PropertyParamFixedCount flag,
// decoders will assume that a property is an array if it has
// either a count parameter or a fixed count other than 1. The
// PropertyParamFixedCount flag allows for fixed-count arrays
with
// one element to be properly decoded as arrays.
bool isArray =
1 != arrayCount ||
0 != (epi.Flags & (PropertyParamCount |
PropertyParamFixedCount));
if (isArray)
{
    wprintf(L" Array[%u]\n", arrayCount);
}

PEVENT_MAP_INFO pMapInfo = nullptr;

// Treat non-array properties as arrays with one element.
for (unsigned arrayIndex = 0; arrayIndex != arrayCount;
arrayIndex += 1)
{
    if (isArray)
    {
        // Print a name for the array element.
        PrintIndent();
        wprintf(L"%ls[%lu]:",
epi.NameOffset ? TeiString(epi.NameOffset) : L"
(noname)",
arrayIndex);
    }

    if (epi.Flags & PropertyStruct)
    {
        // If this property is a struct, recurse and print the
child
        // properties.
        wprintf(L"\n");
        PrintProperties(
            epi.structType.StructstartIndex,
            epi.structType.StructstartIndex +
epi.structType.NumOfStructMembers);
        continue;
    }
}

```

```

        }

        // If the property has an associated map (i.e. an enumerated
        type),
        // try to look up the map data. (If this is an array, we
        only need
        // to do the lookup on the first iteration.)
        if (epi.nonStructType.MapNameOffset != 0 && arrayIndex == 0)
        {
            switch (epi.nonStructType.InType)
            {
                case TDH_INTYPE_UINT8:
                case TDH_INTYPE_UINT16:
                case TDH_INTYPE_UINT32:
                case TDH_INTYPE_HEXINT32:
                    if (m_mapBuffer.size() == 0)
                    {
                        m_mapBuffer.resize(sizeof(EVENT_MAP_INFO));
                    }

                    for (;;)
                    {
                        ULONG cbBuffer = static_cast<ULONG>
(m_mapBuffer.size());
                        ULONG status = TdhGetEventMapInformation(
                            m_pEvent,
                            const_cast<LPWSTR>
(TeiString(epi.nonStructType.MapNameOffset)),
                            reinterpret_cast<PEVENT_MAP_INFO>
(m_mapBuffer.data()),
                            &cbBuffer);

                        if (status == ERROR_INSUFFICIENT_BUFFER &&
                            m_mapBuffer.size() < cbBuffer)
                        {
                            m_mapBuffer.resize(cbBuffer);
                            continue;
                        }
                        else if (status == ERROR_SUCCESS)
                        {
                            pMapInfo = reinterpret_cast<PEVENT_MAP_INFO>
(m_mapBuffer.data());
                        }
                    }

                    break;
                }
                break;
            }
        }

        bool useMap = pMapInfo != nullptr;

        // Loop because we may need to retry the call to
        TdhFormatProperty.
        for (;;)
    
```

```

    {
        ULONG cbBuffer = static_cast<ULONG>
(m_propertyBuffer.size() * 2);
        USHORT cbUsed = 0;
        ULONG status;

        if (0 == propLength &&
            epi.nonStructType.InType == TDH_INTYPE_NULL)
        {
            // TdhFormatProperty doesn't handle INTYPE_NULL.
            if (m_propertyBuffer.empty())
            {
                m_propertyBuffer.push_back(0);
            }
            m_propertyBuffer[0] = 0;
            status = ERROR_SUCCESS;
        }
        else if (
            0 == propLength &&
            0 != (epi.Flags & (PropertyParamLength |
PropertyParamFixedLength)) &&
            (   epi.nonStructType.InType ==
TDH_INTYPE_UNICODESTRING ||
                epi.nonStructType.InType ==
TDH_INTYPE_ANSISTRING))
        {
            // TdhFormatProperty doesn't handle zero-length
counted strings.
            if (m_propertyBuffer.empty())
            {
                m_propertyBuffer.push_back(0);
            }
            m_propertyBuffer[0] = 0;
            status = ERROR_SUCCESS;
        }
        else
        {
            status = TdhFormatProperty(
                const_cast<TRACE_EVENT_INFO*>(pTei),
                useMap ? pMapInfo : nullptr,
                m_pointerSize,
                epi.nonStructType.InType,
                static_cast<USHORT>(
                    epi.nonStructType.OutType ==
TDH_OUTTYPE_NOPRINT
                    ? TDH_OUTTYPE_NULL
                    : epi.nonStructType.OutType),
                propLength,
                static_cast<USHORT>(m_pbDataEnd - m_pbData),
                const_cast<PBYTE>(m_pbData),
                &cbBuffer,
                m_propertyBuffer.data(),
                &cbUsed);
        }
    }

```

```

        if (status == ERROR_INSUFFICIENT_BUFFER &&
            m_propertyBuffer.size() < cbBuffer / 2)
        {
            // Try again with a bigger buffer.
            m_propertyBuffer.resize(cbBuffer / 2);
            continue;
        }
        else if (status == ERROR_EVT_INVALID_EVENT_DATA &&
useMap)
        {
            // If the value isn't in the map, TdhFormatProperty
treats it
            // as an error instead of just putting the number
in. We'll
            // try again with no map.
            useMap = false;
            continue;
        }
        else if (status != ERROR_SUCCESS)
        {
            wprintf(L" [ERROR:TdhFormatProperty:%lu]\n",
status);
        }
        else
        {
            wprintf(L" %ls\n", m_propertyBuffer.data());
            m_pbData += cbUsed;
        }

        break;
    }
}

m_indentLevel -= 1;
}
}

void PrintGuid(GUID const& g)
{
    wprintf(L"%08x-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x",
g.Data1, g.Data2, g.Data3, g.Data4[0], g.Data4[1], g.Data4[2],
g.Data4[3], g.Data4[4], g.Data4[5], g.Data4[6], g.Data4[7]);
}

void PrintFileTime(FILETIME const& ft)
{
    SYSTEMTIME st = {};
    FileTimeToSystemTime(&ft, &st);
    wprintf(L"%04u-%02u-%02uT%02u:%02u:%02u.%03uZ",
st.wYear,
st.wMonth,
st.wDay,
st.wHour,
st.wMinute,
st.wSecond,

```

```

        st.wMilliseconds);
    }

    void PrintIndent()
    {
        wprintf(L"%*ls", m_indentLevel * 2, L"");
    }

    /*
     Returns true if the current event has the EVENT_HEADER_FLAG_STRING_ONLY
     flag set.
    */
    bool IsStringEvent() const
    {
        return (m_pEvent->EventHeader.Flags & EVENT_HEADER_FLAG_STRING_ONLY)
!= 0;
    }

    /*
     Returns true if the current event has the
     EVENT_HEADER_FLAG_TRACE_MESSAGE
     flag set.
    */
    bool IsWppEvent() const
    {
        return (m_pEvent->EventHeader.Flags &
EVENT_HEADER_FLAG_TRACE_MESSAGE) != 0;
    }

    /*
     Converts a TRACE_EVENT_INFO offset (e.g. TaskNameOffset) into a string.
    */
    _Ret_z_ LPCWSTR TeiString(unsigned offset)
    {
        return reinterpret_cast<LPCWSTR>(m_teibuffer.data() + offset);
    }

private:

    TDH_CONTEXT m_tdhContext[1]; // May contain
    TDH_CONTEXT_WPP_TMFSEARCHPATH.

    BYTE m_tdhContextCount; // 1 if a TMF search path is present.
    BYTE m_pointerSize;
    BYTE m_indentLevel; // How far to indent the output.
    EVENT_RECORD* m_pEvent; // The event we're currently printing.
    BYTE const* m_pbData; // Position of the next byte of event data
    to be consumed.
    BYTE const* m_pbDataEnd; // Position of the end of the event data.
    std::vector<USHORT> m_integerValues; // Stored property values for
    resolving array lengths.
    std::vector<BYTE> m_teibuffer; // Buffer for TRACE_EVENT_INFO data.
    std::vector<wchar_t> m_propertyBuffer; // Buffer for the string returned
    by TdhFormatProperty.
    std::vector<BYTE> m_mapBuffer; // Buffer for the data returned by
    TdhGetEventMapInformation.

```

```

};

/*
Parses and stores the command line options.
*/
struct DecoderSettings
{
    std::vector<LPCWSTR> etlFiles;
    std::vector<LPCWSTR> manFiles;
    std::vector<LPCWSTR> binFiles;
    LPCWSTR szTmfSearchPath;
    bool showUsage;

    DecoderSettings(
        int argc,
        _In_count_(argc) LPWSTR argv[])
        : szTmfSearchPath()
        , showUsage()
    {
        for (int i = 1; i < argc; i += 1)
        {
            LPCWSTR szArg = argv[i];
            if (szArg[0] != L'/' && szArg[0] != L'-')
            {
                etlFiles.push_back(szArg);
            }
            else if (szArg[1] == L'\0' ||
                      (szArg[2] != L'\0' && szArg[2] != L':' && szArg[2] != L'='))
            {
                // Options should be /X, /X:Value, or /X=Value
                wprintf(L"ERROR: Incorrectly-formatted option: %ls\n",
szArg);
                showUsage = true;
            }
            else
            {
                LPCWSTR szArgValue = &szArg[3];
                switch (szArg[1])
                {
                    case L'?':
                    case L'h':
                    case L'H':
                        showUsage = true;
                        break;

                    case L'B':
                    case L'b':
                        binFiles.push_back(szArgValue);
                        break;

                    case L'M':
                    case L'm':
                        manFiles.push_back(szArgValue);
                        break;
                }
            }
        }
    }
};

```

```

        case L'T':
        case L't':
            if (szTmfSearchPath == nullptr)
            {
                szTmfSearchPath = szArgValue;
            }
            else
            {
                wprintf(L"ERROR: TMF search path already set:
%ls\n", szArg);
                showUsage = true;
            }
            break;

        default:
            wprintf(L"ERROR: Unrecognized option: %ls\n", szArg);
            showUsage = true;
            break;
    }
}

if (!showUsage && etlFiles.empty())
{
    wprintf(L"ERROR: No ETL files specified.\n");
    showUsage = true;
}
};

/*
Helper class to automatically close TRACEHANDLEs.
*/
class TraceHandles
{
public:

    ~TraceHandles()
    {
        CloseHandles();
    }

    void CloseHandles()
    {
        while (!handles.empty())
        {
            CloseTrace(handles.back());
            handles.pop_back();
        }
    }

    ULONG OpenTraceW(
        _Inout_ EVENT_TRACE_LOGFILEW* pLogFile)
    {
        ULONG status;

```

```

        handles.reserve(handles.size() + 1);
        TRACEHANDLE handle = ::OpenTraceW(pLogFile);
        if (handle == INVALID_PROCESSTRACE_HANDLE)
        {
            status = GetLastError();
        }
        else
        {
            handles.push_back(handle);
            status = 0;
        }

        return status;
    }

    ULONG ProcessTrace(
        _In_opt_ LPFILETIME pStartTime,
        _In_opt_ LPFILETIME pEndTime)
    {
        return ::ProcessTrace(
            handles.data(),
            static_cast<ULONG>(handles.size()),
            pStartTime,
            pEndTime);
    }

private:
    std::vector<TRACEHANDLE> handles;
};

/*
This function will be used as the EventRecordCallback function in
EVENT_TRACE_LOGFILE.
It expects that the EVENT_TRACE_LOGFILE's Context pointer is set to a
DecoderContext.
*/
static void WINAPI EventRecordCallback(
    _In_ EVENT_RECORD* pEventRecord)
{
    try
    {
        // We expect that the EVENT_TRACE_LOGFILE.Context pointer was set
        // with a
        // pointer to a DecoderContext. ProcessTrace will put the Context
        // value
        // into EVENT_RECORD.UserContext.
        DecoderContext* pContext = static_cast<DecoderContext*>
        (pEventRecord->UserContext);

        // The actual decoding work is done in PrintEventRecord.
        pContext->PrintEventRecord(pEventRecord);
    }
    catch (std::exception const& ex)

```

```

    {
        wprintf(L"\nERROR: %hs\n", ex.what());
    }
}

int __cdecl wmain(int argc, _In_count_(argc) LPWSTR argv[])
{
    int exitCode;

    try
    {
        DecoderSettings settings(argc, argv);
        TraceHandles handles;
        if (settings.showUsage)
        {
            wprintf(L""
                    "\nUsage:"
                    "\n"
                    "\n  TdhFormatProperty_Sample [options] filename1.etl
(filename2.etl...)"
                    "\n"
                    "\nOptions:"
                    "\n"
                    "\n  -m:ManifestFile.man  Load decoding data from a manifest
with TdhLoadManifest."
                    "\n  -b:ResourceFile.dll  Load decoding data from a DLL
with"
                    "\n"
                    "\n  -t:TmfSearchPath      Set the TMF search path for WPP
events."
                    "\n"
                    "\n");
            exitCode = 1;
            goto Done;
        }

        DecoderContext context(settings.szTmfSearchPath);

        for (size_t i = 0; i != settings.manFiles.size(); i += 1)
        {
            exitCode = TdhLoadManifest(const_cast<LPWSTR>
(settings.manFiles[i]));
            if (exitCode != 0)
            {
                wprintf(L"ERROR: TdhLoadManifest error %u for manifest:
%ls\n",
                       exitCode,
                       settings.manFiles[i]);
                goto Done;
            }
        }

        for (size_t i = 0; i != settings.binFiles.size(); i += 1)
        {
            exitCode = TdhLoadManifestFromBinary(const_cast<LPWSTR>

```

```

(settings.binFiles[i]));
    if (exitCode != 0)
    {
        wprintf(L"ERROR: TdhLoadManifestFromBinary error %u for
binary: %ls\n",
               exitCode,
               settings.binFiles[i]);
        goto Done;
    }
}

for (size_t i = 0; i != settings.etlFiles.size(); i += 1)
{
    EVENT_TRACE_LOGFILEW logFile = { const_cast<LPWSTR>
(settings.etlFiles[i]) };
    logFile.ProcessTraceMode = PROCESS_TRACE_MODE_EVENT_RECORD;
    logFile.EventRecordCallback = &EventRecordCallback;
    logFile.Context = &context;

    exitCode = handles.OpenTraceW(&logFile);
    if (exitCode != 0)
    {
        wprintf(L"ERROR: OpenTraceW error %u for file: %ls\n",
               exitCode,
               settings.etlFiles[i]);
        goto Done;
    }

    wprintf(L"Opened: %ls\n", logFile.LogFileName);

    // Optionally print information read from the log file.
    // For example, show information about lost buffers and events.

    if (logFile.LogfileHeader.BuffersLost != 0)
    {
        wprintf(L"  **BuffersLost = %lu\n",
logFile.LogfileHeader.BuffersLost);
    }

    if (logFile.LogfileHeader.EventsLost != 0)
    {
        wprintf(L"  **EventsLost = %lu\n",
logFile.LogfileHeader.EventsLost);
    }
}

exitCode = handles.ProcessTrace(nullptr, nullptr);
if (exitCode != 0)
{
    wprintf(L"ERROR: ProcessTrace error %u\n",
           exitCode);
    goto Done;
}
}

catch (std::exception const& ex)

```

```
{  
    wprintf(L"\nERROR: %hs\n", ex.what());  
    exitCode = 1;  
}
```

Done:

```
    return exitCode;  
}
```

# Using TdhGetProperty to Consume Event Data

Article • 08/08/2022 • 21 minutes to read

The following example shows how to consume event data using the [TdhGetProperty](#) function.

## ⓘ Important

TdhGetProperty is not recommended for new code.

- For non-WPP events, extracting property values using TdhGetProperty is very inefficient, as it requires scanning the event's properties each time the TdhGetProperty API is called. Refer to [the TdhFormatProperty sample](#) for a more efficient pattern for reading property values.
- For WPP events, the TdhGetWppProperty and TdhGetWppMessage APIs are more flexible than TdhGetProperty. For example, the TdhGetWpp\* APIs can load WPP decoding information from PDB files. Consider using TdhGetWppProperty or TdhGetWppMessage instead of TdhGetProperty when decoding WPP events.

C++

```
/*
Sample for decoding ETW events using TdhGetProperty.
```

**IMPORTANT:** TdhGetProperty is not recommended for new code.

- Extracting property values from non-WPP events using TdhGetProperty is very inefficient, as it requires scanning the event's properties each time the TdhGetProperty API is called. Refer to [the TdhFormatProperty sample](#) for a more efficient pattern for reading property values.
- The TdhGetWppProperty and TdhGetWppMessage APIs are more efficient and more flexible than TdhGetProperty. For example, the TdhGetWpp\* APIs can load WPP decoding information from PDB files. Consider using TdhGetWppProperty or TdhGetWppMessage instead of TdhGetProperty when decoding WPP events.

This sample demonstrates the following:

- How to process events from ETL files using OpenTrace and ProcessTrace.

```

- How to get TRACE_EVENT_INFO data for an event using
TdhGetEventInformation.
- How to format a WPP message using TdhGetProperty.
- How to extract property values from non-WPP events using TdhGetProperty.
- How to format map (enum) values using the EVENT_MAP_INFO data returned by
TdhGetEventMapInformation.
*/

```

```

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN 1 // Exclude rarely-used APIs from <windows.h>
#endif

#include <windows.h>

#define INITGUID // Ensure that EventTraceGuid is defined.
#include <evntrace.h>
#undef INITGUID

#include <sddl.h> // ConvertSidToStringSid
#include <tdh.h>
#include <vector>

#include <wchar.h> // wprintf
#include <stdlib.h> // _byteswap_ushort

#pragma comment(lib, "tdh.lib") // Link against TDH.dll

// Support building this sample using older versions of the Windows SDK:
#define EventNameOffset           ActivityIDNameOffset
#define EventAttributesOffset      RelatedActivityIDNameOffset
#define TDH_INTYPE_MANIFEST_COUNTEDSTRING static_cast<_TDH_IN_TYPE>(22)
#define TDH_INTYPE_MANIFEST_COUNTEDANSISTRING static_cast<_TDH_IN_TYPE>(23)
#define TDH_INTYPE_MANIFEST_COUNTEDBINARY   static_cast<_TDH_IN_TYPE>(25)
#define TDH_OUTTYPE_CODE_POINTER    static_cast<_TDH_OUT_TYPE>(37)

/*
Decodes event data using TdhGetEventInformation and TdhGetProperty. Prints
the
event information to stdout.

```

We use a context object so we can reuse buffers instead of allocating new buffers and freeing them for each event.

```

    if (szTmfSearchPath != nullptr)
    {
        p->ParameterValue = reinterpret_cast<UINT_PTR>(szTmfSearchPath);
        p->ParameterType = TDH_CONTEXT_WPP_TMFSEARCHPATH;
        p->ParameterSize = 0;
        p += 1;
    }

    m_tdhContextCount = static_cast<BYTE>(p - m_tdhContext);
}

/*
Decode and print the data for an event.
Might throw an exception for out-of-memory conditions. Caller should
catch
the exception before returning from the ProcessTrace callback.
*/
void PrintEventRecord(
    _In_ EVENT_RECORD* pEventRecord)
{
    if (pEventRecord->EventHeader.EventDescriptor.Opcode ==
EVENT_TRACE_TYPE_INFO &&
        pEventRecord->EventHeader.ProviderId == EventTraceGuid)
    {
        /*
        The first event in every ETL file contains the data from the
file header.
        This is the same data as was returned in the
EVENT_TRACE_LOGFILEW by
        OpenTrace. Since we've already seen this information, we'll skip
this
        event.
    */
        return;
    }

    // Reset state to process a new event.
    m_indentLevel = 1;
    m_pEvent = pEventRecord;
    m_pdd.clear();

    // There is a lot of information available in the event even without
decoding,
    // including timestamp, PID, TID, provider ID, activity ID, and the
raw data.

    // Show the event timestamp.
    PrintFileTime(reinterpret_cast<FILETIME const&>(m_pEvent-
>EventHeader.TimeStamp));

    if (IsWppEvent())
    {
        // The PrintEvent method works correctly for WPP events, but it
        // generates fairly verbose output. Treat WPP events as a

```

```

special
    // case and generate more compact output for them.
    PrintWppEvent();
}
else
{
    PrintEvent();
}
}

private:

/*
Print the primary properties for a WPP event.
*/
void PrintWppEvent()
{
    /*
TDH supports a set of known properties for WPP events:
- "Version": UINT32 (usually 0)
- "TraceGuid": GUID
- "GuidName": UNICODESTRING (module name)
- "GuidTypeName": UNICODESTRING (source file name and line number)
- "ThreadId": UINT32
- "SystemTime": SYSTEMTIME
- "UserTime": UINT32
- "KernelTime": UINT32
- "SequenceNum": UINT32
- "ProcessId": UINT32
- "CpuNumber": UINT32
- "Indent": UINT32
- "FlagsName": UNICODESTRING
- "LevelName": UNICODESTRING
- "FunctionName": UNICODESTRING
- "ComponentName": UNICODESTRING
- "SubComponentName": UNICODESTRING
- "FormattedString": UNICODESTRING
- "RawSystemTime": FILETIME
- "ProviderGuid": GUID (usually 0)
*/
}

// Since WPP events always have the same properties, there is no
need
    // to query TdhGetEventInformation to find the list of properties.
    // We'll just use TdhGetProperty to get the properties we need.
    wprintf(L" ");
    PrintWppStringProperty(L"GuidName"); // Module name (WPP's
"CurrentDir" variable)
    wprintf(L" ");
    PrintWppStringProperty(L"GuidTypeName"); // Source code file name +
line number
    wprintf(L" ");
    PrintWppStringProperty(L"FunctionName");
    wprintf(L"\n");
    PrintIndent();
}

```

```

        PrintWppStringProperty(L"FormattedMessage");
        wprintf(L"\n");
    }

/*
Print the value of the given UNICODESTRING property.
*/
void PrintWppStringProperty(_In_z_ LPCWSTR szPropertyName)
{
    // Assume that the property name corresponds to a UNICODESTRING
    // property.
    // Assume that the PROPERTY_DATA_DESCRIPTOR stack is empty.

    // Put the property name onto the stack.
    PushPdd(szPropertyName);

    // Call TdhGetProperty using a the current PROPERTY_DATA_DESCRIPTOR
    // stack (should have just one item on the stack right now). If
    // successful, put the property data into m_propertyBuffer.
    ULONG status = GetProperty();

    // Clear the PROPERTY_DATA_DESCRIPTOR stack.
    PopPdd();

    if (status != ERROR_SUCCESS)
    {
        wprintf(L"[TdhGetProperty(%ls) error %u]", szPropertyName,
status);
    }
    else
    {
        // Print the FormattedString property data (nul-terminated
        // wchar_t string).
        wprintf(L"%ls", reinterpret_cast<LPCWSTR>
(m_propertyBuffer.data()));
    }
}

/*
Use TdhGetEventInformation to obtain information about this event
(including the names and types of the event's properties). Print some
basic information about the event (provider name, event name), then
print
each property (using TdhGetProperty to read each property value).
*/
void PrintEvent()
{
    ULONG status;
    ULONG cb;

    // Try to get event decoding information from TDH.
    cb = static_cast<ULONG>(m_teibuffer.size());
    status = TdhGetEventInformation(
        m_pEvent,
        m_tdhContextCount,

```

```

    m_tdhContextCount ? m_tdhContext : nullptr,
    reinterpret_cast<TRACE_EVENT_INFO*>(m_teibuffer.data()),
    &cb);
if (status == ERROR_INSUFFICIENT_BUFFER)
{
    m_teibuffer.resize(cb);
    status = TdhGetEventInformation(
        m_pEvent,
        m_tdhContextCount,
        m_tdhContextCount ? m_tdhContext : nullptr,
        reinterpret_cast<TRACE_EVENT_INFO*>(m_teibuffer.data()),
        &cb);
}

if (status != ERROR_SUCCESS)
{
    // TdhGetEventInformation failed so there isn't a lot we can do.
    // The provider ID might be helpful in tracking down the right
    // manifest or TMF path.
    wprintf(L" ");
    PrintGuid(m_pEvent->EventHeader.ProviderId);
    wprintf(L"\n");
}
else
{
    // TDH found decoding information. Print some basic info about
the event,
    // then format the event contents.

    TRACE_EVENT_INFO const* const pTei =
        reinterpret_cast<TRACE_EVENT_INFO const*>
(m_teibuffer.data());

    if (pTei->ProviderNameOffset != 0)
    {
        // Event has a provider name -- show it.
        wprintf(L" %ls", TeiString(pTei->ProviderNameOffset));
    }
    else
    {
        // No provider name so print the provider ID.
        wprintf(L" ");
        PrintGuid(m_pEvent->EventHeader.ProviderId);
    }

    // Show core important event properties - try to show some kind
of "event name".
    if (pTei->DecodingSource == DecodingSourceWbem ||
        pTei->DecodingSource == DecodingSourceWPP)
    {
        // OpcodeName is usually the best "event name" property for
WBEM/WPP events.
        if (pTei->OpcodeNameOffset != 0)
        {
            wprintf(L" %ls", TeiString(pTei->OpcodeNameOffset));
        }
    }
}

```

```

        }

        wprintf(L"\n");
    }
    else
    {
        if (pTei->EventNameOffset != 0)
        {
            // Event has an EventName, so print it.
            wprintf(L" %ls", TeiString(pTei->EventNameOffset));
        }
        else if (pTei->TaskNameOffset != 0)
        {
            // EventName is a recent addition, so not all events
have it.
            // Many events use TaskName as an event identifier, so
print it if present.
            wprintf(L" %ls", TeiString(pTei->TaskNameOffset));
        }

        wprintf(L"\n");

        // Show EventAttributes if available.
        if (pTei->EventAttributesOffset != 0)
        {
            PrintIndent();
            wprintf(L"EventAttributes: %ls\n", TeiString(pTei-
>EventAttributesOffset));
        }
    }

    if (IsStringEvent())
    {
        // The event was written using EventWriteString.
        // We'll handle it later.
    }
    else
    {
        // The event is a MOF, manifest, or TraceLogging event.

        // To help resolve PropertyParamCount and
PropertyParamLength,
        // we will record the values of all integer properties as we
        // reach them. Before we start, clear out any old values and
        // resize the vector with room for the new values.
        m_integerValues.clear();
        m_integerValues.resize(pTei->PropertyCount);

        // Recursively print the event's properties.
        PrintProperties(0, pTei->TopLevelPropertyCount);
    }
}

if (IsStringEvent())
{

```

```

        // The event was written using EventWriteString.
        // We can print it whether or not we have decoding information.
        LPCWSTR pchData = static_cast<LPCWSTR>(m_pEvent->UserData);
        unsigned cchData = m_pEvent->UserDataLength / 2;
        PrintIndent();

        // It's probably nul-terminated, but just in case, limit to
        cchData chars.
        wprintf(L"%.*ls\n", cchData, pchData);
    }
}

/*
Prints out the values of properties from begin..end.
Called by PrintEventRecord for the top-level properties.
If there are structures, this will be called recursively for the child
properties.
*/
void PrintProperties(unsigned propBegin, unsigned propEnd)
{
    TRACE_EVENT_INFO const* const pTei =
        reinterpret_cast<TRACE_EVENT_INFO const*>(m_teibuffer.data());

    for (unsigned propIndex = propBegin; propIndex != propEnd; propIndex
+= 1)
    {
        EVENT_PROPERTY_INFO const& epi = pTei-
>EventPropertyInfoArray[propIndex];

        PrintIndent();

        // Print the property's name.
        wprintf(L"%ls:", epi.NameOffset ? TeiString(epi.NameOffset) : L"
(noname)");

        // The array of PROPERTY_DATA_DESCRIPTORs that we pass to
        // TdhGetProperty makes a path that describes the property
        // we want to access. If the property is a top-level property,
        // we only need one PROPERTY_DATA_DESCRIPTOR to identify it.
        // If we want to access a child property (nested within a
        // structure), we need to provide a full path from the top-
        // level property to the child property, with one
        // PROPERTY_DATA_DESCRIPTOR for each level of nesting.
        // To create this path, we have a stack of
        // PROPERTY_DATA_DESCRIPTORs.

        // Add the property to the stack of PROPERTY_DATA_DESCRIPTORs
        // that will be used when we call TdhGetProperty. For top-level
        // properties, there will be only one PROPERTY_DATA_DESCRIPTOR
in
        // the stack, but for child properties there may be two or more.
        unsigned const pddIndex = PushPdd(epi.NameOffset ?
TeiString(epi.NameOffset) : L"");
        m_indentLevel += 1;
}

```

```

        // We recorded the values of all previous integer properties
just
        // in case we need to determine the array count for a variable-
size
        // array (PropertyParamCount).
USHORT const arrayCount =
    (epi.Flags & PropertyParamCount)
    ? m_integerValues[epi.countPropertyIndex] // Look up the
value of a previous property
    : epi.count;

        // Note that PropertyParamFixedCount is a new flag and is
ignored
        // by many decoders. Without the PropertyParamFixedCount flag,
// decoders will assume that a property is an array if it has
// either a count parameter or a fixed count other than 1. The
// PropertyParamFixedCount flag allows for fixed-count arrays
with
        // one element to be properly decoded as arrays.
bool isArray =
    1 != arrayCount ||
    0 != (epi.Flags & (PropertyParamCount |
PropertyParamFixedCount));
    if (isArray)
    {
        wprintf(L" Array[%u]\n", arrayCount);
    }

        // Treat non-array properties as arrays with one element.
        for (unsigned arrayIndex = 0; arrayIndex != arrayCount;
arrayIndex += 1)
        {
            m_pdd[pddIndex].ArrayIndex = arrayIndex;

            if (isArray)
            {
                // Print a name for the array element.
                PrintIndent();
                wprintf(L"%ls[%lu]:",
                    epi.NameOffset ? TeiString(epi.NameOffset) : L"
(noname)",
                    arrayIndex);
            }

            if (epi.Flags & PropertyStruct)
            {
                // If this property is a struct, recurse and print the
child
                // properties.
                wprintf(L"\n");
                PrintProperties(
                    epi.structType.StructstartIndex,
                    epi.structType.StructstartIndex +
epi.structType.NumOfStructMembers);
                continue;
            }
        }
    }
}

```

```

        }

        // Call TdhGetProperty using the PROPERTY_DATA_DESCRIPTORs
that are
        // currently on our PROPERTY_DATA_DESCRIPTOR stack. If
successful,
        // put the property data into m_propertyBuffer.
        ULONG status = GetProperty();
        if (status != ERROR_SUCCESS)
        {
            wprintf(L" TdhGetProperty error %u\n", status);
            continue;
        }

        if (!isArray)
        {
            // If the property is an integer, save it in case we
need it later
            // to resolve a PropertyParamCount reference.
            USHORT& integerValue = m_integerValues[propIndex];
            void const* pData = m_propertyBuffer.data();
            switch (epi.nonStructType.InType)
            {
                case TDH_INTYPE_UINT8:
                case TDH_INTYPE_INT8:
                    integerValue = *static_cast<UINT8 const*>(pData);
                    break;
                case TDH_INTYPE_INT16:
                case TDH_INTYPE_UINT16:
                case TDH_INTYPE_INT32:
                case TDH_INTYPE_UINT32:
                case TDH_INTYPE_HEXINT32:
                    integerValue = *static_cast<UINT16 const*>(pData);
                    break;
            }
        }

        wprintf(L" ");

        // Print the data in m_propertyBuffer.
        PrintPropertyValue(
            epi.nonStructType.InType,
            epi.nonStructType.OutType,
            epi.nonStructType.MapNameOffset);
        wprintf(L"\n");
    }

    PopPdd();
    m_indentLevel -= 1;
}
}

/*
Prints the data in m_propertyBuffer.

```

For details on valid intype and outtype values, refer to winmeta.xml and tdh.h.

This function assumes that `m_propertyBuffer` is correctly-sized for the data type. For example, it assumes that when `inType` is `UINT32`, `m_propertyBuffer.size() == 4`.

```
/*
void PrintPropertyValue(USHORT inType, USHORT outType, unsigned
mapNameOffset)
{
    void* const pData = m_propertyBuffer.data();
    unsigned const cData = static_cast<unsigned>
(m_propertyBuffer.size());

    switch (inType)
    {
        case TDH_INTYPE_UNICODESTRING:
        case TDH_INTYPE_NONNULLTERMINATEDSTRING: // WBEM, deprecated

        /*
        Note that the UNICODESTRING type may or may not be nul-
terminated.
        If the property has a non-zero length, has a PropertyParamLength
        flag, or has a PropertyParamFixedLength flag, then the length is
        the length of the string (in characters). Otherwise, the string
        is
        nul-terminated. This sample does not distinguish between these
        cases and prints to the end of the buffer or the nul, whichever
        comes first. (This behavior also works well for the
        NONNULLTERMINATEDSTRING type.)
    */

    // Valid outtypes: string, xml, json
    wprintf(L"%.*ls", cData / 2, static_cast<LPCWSTR>(pData));
    break;

    case TDH_INTYPE_ANSISTRING:
    case TDH_INTYPE_NONNULLTERMINATEDANSISTRING: // WBEM, deprecated

    /*
    Like UNICODESTRING, the ANSISTRING type may or may not be
    nul-terminated based on the flags and length.

    This sample does not distinguish between the possible encodings
    used by the ANSISTRING data types. In particular, data using the
    xml, json, or utf8 outtype should be treated as utf-8, while
    data
    using the string or default outtype is treated as using an
    unspecified code page (typically the decoding machine's default
    code page is used).
*/
    */

    // Valid outtypes: string, xml, json, utf8
    wprintf(L"%.*hs", cData, static_cast<LPCSTR>(pData));
    break;
}
```

```

case TDH_INTYPE_INT8:

    // Valid outtypes: byte, string
    switch (outType)
    {
        default:
            wprintf(L"%i", *static_cast<signed char*>(pData));
            break;
        case TDH_OUTTYPE_STRING:
            wprintf(L"%c", *static_cast<unsigned char*>(pData));
            break;
    }
    break;

case TDH_INTYPE_UINT8:

    // UINT8 properties might have an associated map.
    if (mapNameOffset != 0 &&
        PrintMapString(mapNameOffset, *static_cast<unsigned char*>
(pData)))
    {
        break;
    }

    // Valid outtypes: unsignedByte, HexInt8, boolean, string
    switch (outType)
    {
        default: // treat boolean as integer
            wprintf(L"%u", *static_cast<unsigned char*>(pData));
            break;
        case TDH_OUTTYPE_HEXINT8:
            wprintf(L"0x%x", *static_cast<unsigned char*>(pData));
            break;
        case TDH_OUTTYPE_STRING:
            wprintf(L"%hc", *static_cast<unsigned char*>(pData));
            break;
    }
    break;

case TDH_INTYPE_INT16:

    // Valid outtypes: short
    wprintf(L"%i", *static_cast<short*>(pData));
    break;

case TDH_INTYPE_UINT16:

    // UINT16 properties might have an associated map.
    if (mapNameOffset != 0 &&
        PrintMapString(mapNameOffset, *static_cast<unsigned short*>
(pData)))
    {
        break;
    }

```

```

// Valid outtypes: unsignedShort, port, HexInt16, string
switch (outType)
{
default:
    wprintf(L"%u", *static_cast<unsigned short*>(pData));
    break;
case TDH_OUTTYPE_PORT:
    wprintf(L"%u", _byteswap_ushort(*static_cast<unsigned short*>(pData)));
    break;
case TDH_OUTTYPE_HEXINT16:
    wprintf(L"0x%x", *static_cast<unsigned short*>(pData));
    break;
case TDH_OUTTYPE_STRING:
    wprintf(L"%lc", *static_cast<unsigned short*>(pData));
    break;
}
break;

case TDH_INTYPE_INT32:

// Valid outtypes: int, hresult
switch (outType)
{
default:
    wprintf(L"%i", *static_cast<int*>(pData));
    break;
case TDH_OUTTYPE_HRESULT:
    wprintf(L"0x%x", *reinterpret_cast<unsigned*>
(m_propertyBuffer.data()));
    break;
}
break;

case TDH_INTYPE_UINT32:

// UINT32 properties might have an associated map.
if (mapNameOffset != 0 &&
    PrintMapString(mapNameOffset, *static_cast<unsigned*>
(pData)))
{
    break;
}

// Valid outtypes: unsignedInt, PID, TID, IPv4, ETWTIME,
ErrorCode, Win32Error,
// NTSTATUS, HexInt32, CodePointer
switch (outType)
{
default:
    wprintf(L"%u", *static_cast<unsigned*>(pData));
    break;
case TDH_OUTTYPE_HEXINT32:
case TDH_OUTTYPE_ERRORCODE:

```

```
case TDH_OUTTYPE_HRESULT:
case TDH_OUTTYPE_NTSTATUS:
case TDH_OUTTYPE_CODE_POINTER:
    wprintf(L"0x%08x", *static_cast<unsigned*>(pData));
    break;
case TDH_OUTTYPE_IPV4:
    wprintf(L"%u.%u.%u.%u",
           static_cast<BYTE*>(pData)[0],
           static_cast<BYTE*>(pData)[1],
           static_cast<BYTE*>(pData)[2],
           static_cast<BYTE*>(pData)[3]);
    break;
}
break;

case TDH_INTYPE_INT64:
    // Valid outtypes: long
    wprintf(L"%lli", *static_cast<__int64*>(pData));
    break;

case TDH_INTYPE_UINT64:
    // Valid outtypes: unsignedLong, ETWTIME, HexInt64, CodePointer
    switch (outType)
    {
    default:
        wprintf(L"%llu", *static_cast<unsigned __int64*>(pData));
        break;
    case TDH_OUTTYPE_HEXINT64:
    case TDH_OUTTYPE_CODE_POINTER:
        wprintf(L"0x%llx", *static_cast<unsigned __int64*>(pData));
        break;
    }
    break;

case TDH_INTYPE_FLOAT:
    // Valid outtypes: float
    wprintf(L"%f", *static_cast<float*>(pData));
    break;

case TDH_INTYPE_DOUBLE:
    // Valid outtypes: double
    wprintf(L"%f", *static_cast<double*>(pData));
    break;

case TDH_INTYPE_BOOLEAN:
    // Valid outtypes: boolean
    wprintf(L"%u", *static_cast<unsigned*>(pData)); // 4-byte BOOL
type
    break;
```

```
case TDH_INTYPE_BINARY:

    // Valid outtypes: hexBinary, IPv6, SocketAddress,
Pkcs7WithTypeInfo
    switch (outType)
    {
    default:
        PrintHexBinary(pData, cData);
        break;
    case TDH_OUTTYPE_IPV6:
        PrintIPv6(pData, cData);
        break;
    }
    break;

case TDH_INTYPE_GUID:

    // Valid outtypes: guid
PrintGuid(*static_cast<GUID*>(pData));
break;

case TDH_INTYPE_POINTER:

    // Valid outtypes: HexInt64, CodePointer, long, unsignedLong
    // Assume that TdhGetProperty has correctly determined size.
    switch (outType)
    {
    default:
        if (cData == 8)
        {
            wprintf(L"0x%llx", *static_cast<unsigned __int64*>
(pData));
        }
        else
        {
            wprintf(L"0x%lx", *static_cast<unsigned*>(pData));
        }
        break;
    case TDH_OUTTYPE_LONG:
        if (cData == 8)
        {
            wprintf(L"%lli", *static_cast<__int64*>(pData));
        }
        else
        {
            wprintf(L"%i", *static_cast<int*>(pData));
        }
        break;
    case TDH_OUTTYPE_UNSIGNEDLONG:
        if (cData == 8)
        {
            wprintf(L"%llu", *static_cast<unsigned __int64*>
(pData));
        }
        else
```

```

        {
            wprintf(L"%u", *static_cast<unsigned*>(pData));
        }
        break;
    }
    break;

case TDH_INTYPE_FILETIME:

    // Valid outtypes: dateTime, DateTimeCultureInsensitive
    PrintFileTime(*static_cast<FILETIME*>(pData));
    break;

case TDH_INTYPE_SYSTEMTIME:

    // Valid outtypes: dateTime, DateTimeCultureInsensitive
    PrintSystemTime(*static_cast<SYSTEMTIME*>(pData));
    break;

case TDH_INTYPE_SID:

    // Valid outtypes: string
    PrintSid(pData, cData);
    break;

case TDH_INTYPE_HEXINT32:

    // HEXINT32 properties might have an associated map.
    if (mapNameOffset != 0 &&
        PrintMapString(mapNameOffset, *static_cast<unsigned*>
(pData)))
    {
        break;
    }

    // Valid outtypes: HexInt32, ErrorCode, Win32Error, NTSTATUS,
CodePointer
    wprintf(L"0x%x", *static_cast<unsigned*>(pData));
    break;

case TDH_INTYPE_HEXINT64:

    // Valid outtypes: HexInt64, CodePointer
    wprintf(L"0x%llx", *static_cast<unsigned __int64*>(pData));
    break;

case TDH_INTYPE_MANIFEST_COUNTEDSTRING: // Added in Windows 10 2018
Fall update
    case TDH_INTYPE_COUNTEDSTRING: // WBEM
    case TDH_INTYPE_REVERSEDCOUNTEDSTRING: // WBEM, deprecated

        // Valid outtypes: string, xml, json
        if (cData >= 2) // Assume first 2 bytes are bytecount.
        {
            wprintf(L"%.*ls", (cData / 2) - 1, static_cast<LPCWSTR>

```

```
(pData) + 1);
    }
    break;

    case TDH_INTYPE_MANIFEST_COUNTEDANSISTRING: // Added in Windows 10
2018 Fall update
    case TDH_INTYPE_COUNTEDANSISTRING: // WBEM
    case TDH_INTYPE_REVERSEDCOUNTEDANSISTRING: // WBEM, deprecated

        // Valid outtypes: string, xml, json, utf8
        if (cData >= 2) // Assume first 2 bytes are bytecount.
        {
            wprintf(L"%.*hs", cData - 2, static_cast<LPCSTR>(pData) +
2);
        }
        break;

    case TDH_INTYPE_MANIFEST_COUNTEDBINARY: // Added in Windows 10 2018
Fall update

        // Valid outtypes: hexBinary, IPv6, SocketAddress,
Pkcs7WithTypeInfo
        if (cData >= 2) // Assume first 2 bytes are bytecount.
        {
            switch (outType)
            {
                default:
                    PrintHexBinary(static_cast<BYTE*>(pData) + 2, cData -
2);
                    break;
                case TDH_OUTTYPE_IPV6:
                    PrintIPv6(static_cast<BYTE*>(pData) + 2, cData - 2);
                    break;
            }
        }
        break;

    case TDH_INTYPE_UNICODECHAR: // WBEM

        wprintf(L"%lc", *static_cast<unsigned short*>(pData));
        break;

    case TDH_INTYPE_ANSICCHAR: // WBEM

        wprintf(L"%hc", *static_cast<unsigned char*>(pData));
        break;

    case TDH_INTYPE_SIZE_T: // WBEM

        if (cData == 8)
        {
            wprintf(L"%llu", *static_cast<unsigned __int64*>(pData));
        }
        else
        {
```

```

        wprintf(L"%u", *static_cast<unsigned*>(pData));
    }
    break;

    case TDH_INTYPE_HEXDUMP: // WBEM, deprecated

        if (cData >= 4) // Assume first 4 bytes are bytecount.
        {
            PrintHexBinary(static_cast<BYTE*>(pData) + 4, cData - 4);
        }
        break;

    case TDH_INTYPE_WBEMSID: // WBEM, deprecated
    {
        // A WBEM SID is actually a TOKEN_USER structure followed
        // by the SID. The TOKEN_USER structure is 2 * pointerSize.
        unsigned const cbTokenUser =
            m_pEvent->EventHeader.Flags &
        EVENT_HEADER_FLAG_32_BIT_HEADER
            ? 2 * 4
            : m_pEvent->EventHeader.Flags &
        EVENT_HEADER_FLAG_64_BIT_HEADER
            ? 2 * 8
            : 2 * sizeof(void*);
        if (cData > cbTokenUser)
        {
            // Skip the TOKEN_USER.
            PrintSid(static_cast<BYTE*>(pData) + cbTokenUser, cData -
cbTokenUser);
        }
        else
        {
            PrintHexBinary(pData, cData);
        }
        break;
    }
    default:

        // Unknown intype. Treat as binary.
        wprintf(L"InType=%u ", inType);
        PrintHexBinary(pData, cData);
        break;
    }
}

bool PrintMapString(ULONG mapNameOffset, ULONG value)
{
    bool matched = false;

    ULONG cb = static_cast<ULONG>(m_mapBuffer.size());
    ULONG status = TdhGetEventMapInformation(
        m_pEvent,
        const_cast<LPWSTR>(TeiString(mapNameOffset)),
        reinterpret_cast<EVENT_MAP_INFO*>(m_mapBuffer.data()),
        &cb);
}

```

```

    if (status == ERROR_INSUFFICIENT_BUFFER)
    {
        m_mapBuffer.reserve(cb);
        m_mapBuffer.resize(m_mapBuffer.capacity());
        status = TdhGetEventMapInformation(
            m_pEvent,
            const_cast<LPWSTR>(TeiString(mapNameOffset)),
            reinterpret_cast<EVENT_MAP_INFO*>(m_mapBuffer.data()),
            &cb);
    }

    if (status == ERROR_SUCCESS)
    {
        EVENT_MAP_INFO const* pMapInfo =
        reinterpret_cast<EVENT_MAP_INFO*>(m_mapBuffer.data());

        if ((pMapInfo->Flag & EVENTMAP_INFO_FLAG_MANIFEST_VALUEMAP) != 0
        ||
            ((pMapInfo->Flag & EVENTMAP_INFO_FLAG_WBEM_VALUEMAP) != 0 &&
            (pMapInfo->Flag & ~EVENTMAP_INFO_FLAG_WBEM_VALUEMAP) !=
            EVENTMAP_INFO_FLAG_WBEM_FLAG))
        {
            if ((pMapInfo->Flag & EVENTMAP_INFO_FLAG_WBEM_NO_MAP) == 0)
            {
                for (ULONG i = 0; i != pMapInfo->EntryCount; i += 1)
                {
                    if (pMapInfo->MapEntryArray[i].Value == value)
                    {
                        wprintf(L"%ls", MapString(pMapInfo, i));
                        matched = true;
                        break;
                    }
                }
            }
            else if (value < pMapInfo->EntryCount)
            {
                wprintf(L"%ls", MapString(pMapInfo, value));
                matched = true;
            }
        }
        else if (
            (pMapInfo->Flag & EVENTMAP_INFO_FLAG_MANIFEST_BITMAP) != 0
        ||
            (pMapInfo->Flag & EVENTMAP_INFO_FLAG_WBEM_BITMAP) != 0 ||
            ((pMapInfo->Flag & EVENTMAP_INFO_FLAG_WBEM_VALUEMAP) != 0 &&
            (pMapInfo->Flag & ~EVENTMAP_INFO_FLAG_WBEM_VALUEMAP) ==
            EVENTMAP_INFO_FLAG_WBEM_FLAG))
        {
            ULONG matchedBits = 0;
            if ((pMapInfo->Flag & EVENTMAP_INFO_FLAG_WBEM_NO_MAP) == 0)
            {
                for (ULONG i = 0; i != pMapInfo->EntryCount; i += 1)
                {
                    ULONG const mask = pMapInfo->MapEntryArray[i].Value;
                    if ((value & mask) == mask)
                }
            }
        }
    }
}

```

```

        {
            if (mask != 0 || value == 0)
            {
                wprintf(L"%hs%ls", matched ? " | " : "",

MapString(pMapInfo, i));
                matched = true;
                matchedBits |= mask;
            }
        }
    }
else
{
    for (ULONG i = 0; i != pMapInfo->EntryCount; i += 1)
    {
        ULONG const mask = 1 << i;
        if (value & mask)
        {
            wprintf(L"%ls%ls", matched ? L" | " : L"",
MapString(pMapInfo, i));
            matched = true;
            matchedBits |= mask;
        }
    }
}

// Print any unused bits.
if (matched && matchedBits != value)
{
    wprintf(L" | 0x%x", value ^ matchedBits);
}
}

return matched;
}

void PrintGuid(GUID const& g)
{
    wprintf(L"%08x-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x",
g.Data1, g.Data2, g.Data3, g.Data4[0], g.Data4[1], g.Data4[2],
g.Data4[3], g.Data4[4], g.Data4[5], g.Data4[6], g.Data4[7]);
}

void PrintFileTime(FILETIME const& ft)
{
    SYSTEMTIME st = {};
    FileTimeToSystemTime(&ft, &st);
    PrintSystemTime(st);
}

void PrintSystemTime(SYSTEMTIME const& st)
{
    wprintf(L"%04u-%02u-%02uT%02u:%02u:%02u.%03uZ",
st.wYear,

```

```

        st.wMonth,
        st.wDay,
        st.wHour,
        st.wMinute,
        st.wSecond,
        st.wMilliseconds);
    }

void PrintIPv6(_In_bytecount_(cb) void const* p, unsigned cb)
{
    if (cb == 16)
    {
        // Simple (greedy, non-canonical) formatting for IPv6 addresses.

        UINT16 const* const pw = static_cast<UINT16 const*>(p);
        unsigned i;

        // Initial state: we haven't seen a 0:0 yet.

        for (i = 0;; i += 1)
        {
            if (i == 7)
            {
                // No 0:0 in the address.
                wprintf(L"%x", _byteswap_ushort(pw[i]));
                return;
            }

            if (pw[i] == 0 && pw[i + 1] == 0)
            {
                // Found a 0:0
                if (i == 0)
                {
                    wprintf(L":");
                }

                i += 2;
                break;
            }

            wprintf(L"%x:", _byteswap_ushort(pw[i]));
        }

        // We've seen 0:0 and are omitting groups. Scan for a non-zero.

        for (;;)
        {
            if (i == 8)
            {
                // Address ends with 0:0
                wprintf(L":");
                return;
            }

            if (pw[i] != 0)

```

```

        {
            break;
        }

        i += 1;
    }

    for ( ; i != 8; i += 1)
    {
        wprintf(L":%x", _byteswap_ushort(pw[i]));
    }
}
else
{
    PrintHexBinary(p, cb);
}
}

void PrintHexBinary(_In_bytecount_(cb) void const* p, unsigned cb)
{
    BYTE const* pb = static_cast<BYTE const*>(p);
    wprintf(L"0x");
    for (unsigned i = 0; i != cb; i += 1)
    {
        wprintf(L"%02x", pb[i]);
    }
}

void PrintSid(_In_bytecount_(cb) PSID pSid, unsigned cb)
{
    LPWSTR szSid = nullptr;
    if (cb < 8u || // Minimum SID size is 8.
        cb < (8u + static_cast<BYTE*>(pSid)[1] * 4u) || // SID size = 8
+ (4 * SubAuthorityCount)
        !ConvertSidToStringSidW(pSid, &szSid))
    {
        PrintHexBinary(pSid, cb);
    }
    else
    {
        wprintf(L"%ls", szSid);
        LocalFree(szSid);
    }
}

void PrintIndent()
{
    wprintf(L"*%ls", m_indentLevel * 2, L"");
}

/*
Uses the PROPERTY_DATA_DESCRIPTORs in m_pdd (controlled with
PushPdd/PopPdd)
to call TdhGetProperty. If successful, puts the property data into
m_propertyBuffer.

```

```

*/
ULONG GetProperty()
{
    ULONG status;
    ULONG cb = 0;
    status = TdhGetPropertySize(
        m_pEvent,
        m_tdhContextCount,
        m_tdhContextCount ? m_tdhContext : nullptr,
        static_cast<ULONG>(m_pdd.size()),
        m_pdd.data(),
        &cb);
    if (status == ERROR_SUCCESS)
    {
        m_propertyBuffer.resize(cb);
        status = TdhGetProperty(
            m_pEvent,
            m_tdhContextCount,
            m_tdhContextCount ? m_tdhContext : nullptr,
            static_cast<ULONG>(m_pdd.size()),
            m_pdd.data(),
            cb,
            m_propertyBuffer.data());
    }

    return status;
}

/*
Adds a property with the specified name to the PROPERTY_DATA_DESCRIPTOR
stack. Returns the property's index within the m_pdd vector.
*/
unsigned PushPdd(_In_z_ LPCWSTR szPropertyName)
{
    unsigned index = static_cast<unsigned>(m_pdd.size());
    m_pdd.emplace_back();
    PROPERTY_DATA_DESCRIPTOR& pdd = m_pdd.back();
    pdd.PropertyName = reinterpret_cast<UINT_PTR>(szPropertyName);
    pdd.ArrayIndex = 0;
    pdd.Reserved = 0;
    return index;
}

/*
Removes a property from the PROPERTY_DATA_DESCRIPTOR stack.
*/
void PopPdd()
{
    m_pdd.pop_back();
}

/*
Returns true if the current event has the EVENT_HEADER_FLAG_STRING_ONLY
flag set.
*/

```

```

    bool IsStringEvent() const
    {
        return (m_pEvent->EventHeader.Flags & EVENT_HEADER_FLAG_STRING_ONLY)
!= 0;
    }

    /*
     Returns true if the current event has the
EVENT_HEADER_FLAG_TRACE_MESSAGE
flag set.
*/
    bool IsWppEvent() const
    {
        return (m_pEvent->EventHeader.Flags &
EVENT_HEADER_FLAG_TRACE_MESSAGE) != 0;
    }

    /*
Converts a TRACE_EVENT_INFO offset (e.g. TaskNameOffset) into a string.
*/
    _Ret_z_ LPCWSTR TeiString(unsigned offset)
{
    return reinterpret_cast<LPCWSTR>(m_teibuffer.data() + offset);
}

    /*
Converts an EVENT_MAP_INFO MapEntryArray index into a string.
*/
    static _Ret_z_ LPCWSTR MapString(_In_ const EVENT_MAP_INFO* pMapInfo,
unsigned index)
{
    return reinterpret_cast<LPCWSTR>(
        reinterpret_cast<const BYTE*>(pMapInfo) + pMapInfo-
>MapEntryArray[index].OutputOffset);
}

private:

    TDH_CONTEXT m_tdhContext[1]; // May contain
TDH_CONTEXT_WPP_TMFSEARCHPATH.
    BYTE m_tdhContextCount;      // 1 if a TMF search path is present.
    BYTE m_indentLevel;         // How far to indent the output.
    EVENT_RECORD* m_pEvent;     // The event we're currently printing.
    std::vector<PROPERTY_DATA_DESCRIPTOR> m_pdd; // Path to the property
we're currently decoding.
    std::vector<USHORT> m_integerValues; // Stored property values for
resolving array lengths.
    std::vector<BYTE> m_teibuffer; // Buffer for TRACE_EVENT_INFO data.
    std::vector<BYTE> m_propertyBuffer; // Buffer for the data returned by
TdhGetProperty.
    std::vector<BYTE> m_mapBuffer; // Buffer for the data returned by
TdhGetEventMapInformation.
};

/*

```



```

        {
            szTmfSearchPath = szArgValue;
        }
        else
        {
            wprintf(L"ERROR: TMF search path already set:
%ls\n", szArg);
            showUsage = true;
        }
        break;

    default:
        wprintf(L"ERROR: Unrecognized option: %ls\n", szArg);
        showUsage = true;
        break;
    }
}
}

if (!showUsage && etlFiles.empty())
{
    wprintf(L"ERROR: No ETL files specified.\n");
    showUsage = true;
}
};

void CloseHandles()
    {
        while (!handles.empty())
        {
            CloseTrace(handles.back());
            handles.pop_back();
        }
    }

    ULONG OpenTraceW(
        _Inout_ EVENT_TRACE_LOGFILEW* pLogFile)
    {
        ULONG status;

        handles.reserve(handles.size() + 1);
        TRACEHANDLE handle = ::OpenTraceW(pLogFile);

```

```

        if (handle == INVALID_PROCESSTRACE_HANDLE)
        {
            status = GetLastError();
        }
        else
        {
            handles.push_back(handle);
            status = 0;
        }

        return status;
    }

    ULONG ProcessTrace(
        _In_opt_ LPFILETIME pStartTime,
        _In_opt_ LPFILETIME pEndTime)
{
    return ::ProcessTrace(
        handles.data(),
        static_cast<ULONG>(handles.size()),
        pStartTime,
        pEndTime);
}

private:

    std::vector<TRACEHANDLE> handles;
};

/*
This function will be used as the EventRecordCallback function in
EVENT_TRACE_LOGFILE.
It expects that the EVENT_TRACE_LOGFILE's Context pointer is set to a
DecoderContext.
*/
static void WINAPI EventRecordCallback(
    _In_ EVENT_RECORD* pEventRecord)
{
    try
    {
        // We expect that the EVENT_TRACE_LOGFILE.Context pointer was set
        // with a
        // pointer to a DecoderContext. ProcessTrace will put the Context
        // value
        // into EVENT_RECORD.UserContext.
        DecoderContext* pContext = static_cast<DecoderContext*>
        (pEventRecord->UserContext);

        // The actual decoding work is done in PrintEventRecord.
        pContext->PrintEventRecord(pEventRecord);
    }
    catch (std::exception const& ex)
    {
        wprintf(L"\nERROR: %hs\n", ex.what());
    }
}

```

```

}

int __cdecl wmain(int argc, _In_count_(argc) LPWSTR argv[])
{
    int exitCode;

    try
    {
        DecoderSettings settings(argc, argv);
        TraceHandles handles;
        if (settings.showUsage)
        {
            wprintf(L""
                    "\nUsage:"
                    "\n"
                    "\n  TdhGetProperty_Sample [options] filename1.etl
(filename2.etl...)"
                    "\n"
                    "\nOptions:"
                    "\n"
                    "\n  -m:ManifestFile.man  Load decoding data from a manifest
with TdhLoadManifest."
                    "\n  -b:ResourceFile.dll  Load decoding data from a DLL
with"
                    "\n"
                    "\n  -t:TmfSearchPath      Set the TMF search path for WPP
events."
                    "\n"
                    "\n");
            exitCode = 1;
            goto Done;
        }

        DecoderContext context(settings.szTmfSearchPath);

        for (size_t i = 0; i != settings.manFiles.size(); i += 1)
        {
            exitCode = TdhLoadManifest(const_cast<LPWSTR>
(settings.manFiles[i]));
            if (exitCode != 0)
            {
                wprintf(L"ERROR: TdhLoadManifest error %u for manifest:
%ls\n",
                        exitCode,
                        settings.manFiles[i]);
                goto Done;
            }
        }

        for (size_t i = 0; i != settings.binFiles.size(); i += 1)
        {
            exitCode = TdhLoadManifestFromBinary(const_cast<LPWSTR>
(settings.binFiles[i]));
            if (exitCode != 0)
            {

```

```

        wprintf(L"ERROR: TdhLoadManifestFromBinary error %u for
binary: %ls\n",
            exitCode,
            settings.binFiles[i]);
        goto Done;
    }
}

for (size_t i = 0; i != settings.etlFiles.size(); i += 1)
{
    EVENT_TRACE_LOGFILEW logFile = { const_cast<LPWSTR>
(settings.etlFiles[i]) };
    logFile.ProcessTraceMode = PROCESS_TRACE_MODE_EVENT_RECORD;
    logFile.EventRecordCallback = &EventRecordCallback;
    logFile.Context = &context;

    exitCode = handles.OpenTraceW(&logFile);
    if (exitCode != 0)
    {
        wprintf(L"ERROR: OpenTraceW error %u for file: %ls\n",
            exitCode,
            settings.etlFiles[i]);
        goto Done;
    }

    wprintf(L"Opened: %ls\n", logFile.LogFileName);

    // Optionally print information read from the log file.
    // For example, show information about lost buffers and events.

    if (logFile.LogfileHeader.BuffersLost != 0)
    {
        wprintf(L" **BuffersLost = %lu\n",
logFile.LogfileHeader.BuffersLost);
    }

    if (logFile.LogfileHeader.EventsLost != 0)
    {
        wprintf(L" **EventsLost = %lu\n",
logFile.LogfileHeader.EventsLost);
    }
}

exitCode = handles.ProcessTrace(nullptr, nullptr);
if (exitCode != 0)
{
    wprintf(L"ERROR: ProcessTrace error %u\n",
            exitCode);
    goto Done;
}
}

catch (std::exception const& ex)
{
    wprintf(L"\nERROR: %hs\n", ex.what());
    exitCode = 1;
}

```

```
}
```

Done:

```
    return exitCode;  
}
```

# Enumerating Providers

Article • 03/04/2021 • 2 minutes to read

To retrieve a list of providers that have installed their manifest or MOF classes on the computer, call the [TdhEnumerateProviders](#) function.

The following example shows how to enumerate providers.

C++

```
#include <windows.h>
#include <stdio.h>
#include <tdh.h>

#pragma comment(lib, "tdh.lib")

#define MAX_GUID_SIZE 39

void wmain(void)
{
    DWORD status = ERROR_SUCCESS;
    PROVIDER_ENUMERATION_INFO* penum = NULL;      // Buffer that contains
provider information
    PROVIDER_ENUMERATION_INFO* ptemp = NULL;
    DWORD BufferSize = 0;                         // Size of the penum buffer
    HRESULT hr = S_OK;                           // Return value for
StringFromGUID2
    WCHAR StringGuid[MAX_GUID_SIZE];
    DWORD RegisteredMOFCount = 0;
    DWORD RegisteredManifestCount = 0;

    // Retrieve the required buffer size.

    status = TdhEnumerateProviders(penum, &BufferSize);

    // Allocate the required buffer and call TdhEnumerateProviders. The list
of
    // providers can change between the time you retrieved the required
buffer
    // size and the time you enumerated the providers, so call
TdhEnumerateProviders
    // in a loop until the function does not return
    // ERROR_INSUFFICIENT_BUFFER.

    while (ERROR_INSUFFICIENT_BUFFER == status)
    {
        ptemp = (PROVIDER_ENUMERATION_INFO*)realloc(penum, BufferSize);
        if (NULL == ptemp)
        {
            wprintf(L"Allocation failed (size=%lu).\n", BufferSize);
            goto cleanup;
        }
        if (SUCCEEDED(hr))
        {
            StringFromGUID2(&StringGuid[0], ptemp->ProviderGuid);
            printf("Provider name: %s\n", StringGuid);
            if (ptemp->ManifestCount != RegisteredManifestCount)
                printf("Manifest count changed from %d to %d.\n",
ptemp->ManifestCount);
            if (ptemp->MOFCount != RegisteredMOFCount)
                printf("MOF count changed from %d to %d.\n",
ptemp->MOFCount);
            if (ptemp->ManifestCount > 0)
                printf("Manifest GUID: %s\n", StringGuid);
            if (ptemp->MOFCount > 0)
                printf("MOF GUID: %s\n", StringGuid);
        }
        else
            printf("TdhEnumerateProviders failed with error %x.\n", hr);
        BufferSize *= 2;
    }
}

void cleanup()
{
    if (penum)
        free(penum);
}
```

```

    }

    penum = ptemp;
    ptemp = NULL;

    status = TdhEnumerateProviders(penum, &BufferSize);
}

if (ERROR_SUCCESS != status)
{
    wprintf(L"TdhEnumerateProviders failed with %lu.\n", status);
}
else
{
    // Loop through the list of providers and print the provider's name,
GUID,
    // and the source of the information (MOF class or instrumentation
manifest).

    for (DWORD i = 0; i < penum->NumberOfProviders; i++)
    {
        hr = StringFromGUID2(penum-
>TraceProviderInfoArray[i].ProviderGuid, StringGuid, ARRAYSIZE(StringGuid));

        if (FAILED(hr))
        {
            wprintf(L"StringFromGUID2 failed with 0x%x\n", hr);
            goto cleanup;
        }

        wprintf(L"Provider name: %s\nProvider GUID: %s\nSource: %s\n\n",
(PWSTR)((PBYTE)(penum)+penum-
>TraceProviderInfoArray[i].ProviderNameOffset),
StringGuid,
(penum->TraceProviderInfoArray[i].SchemaSource) ? L"WMI MOF
class" : L"XML manifest");

        (penum->TraceProviderInfoArray[i].SchemaSource) ?
RegisteredMOFCount++ : RegisteredManifestCount++;
    }

    wprintf(L"\nThere are %d registered providers; %lu are registered
via MOF class and\n%lu are registered via a manifest.\n",
penum->NumberOfProviders,
RegisteredMOFCount,
RegisteredManifestCount);
}

cleanup:

if (penum)
{
    free(penum);
    penum = NULL;
}

```

```
    }  
}
```

# Retrieving Event Metadata

Article • 01/07/2021 • 5 minutes to read

The following example shows how to use the trace data helper functions to retrieve metadata for each event. Also see the examples included with the [TdhQueryProviderFieldInformation](#) and [TdhEnumerateProviderFieldInformation](#) functions.

C++

```
//Turns the DEFINE_GUID for EventTraceGuid into a const.
#define INITGUID

#include <windows.h>
#include <stdio.h>
#include <comdef.h>
#include <guiddef.h>
#include <wbemidl.h>
#include <wmistr.h>
#include <evntrace.h>
#include <tdh.h>

#pragma comment(lib, "tdh.lib")

#define LOGFILE_PATH L"<FULLPATHTOTHELOGFILE.etl>"

static const GUID GUID_NULL =
{ 0x00000000, 0x0000, 0x0000, { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00 } };

// Strings that represent the source of the event metadata.

WCHAR* pSource[] = {L"XML instrumentation manifest", L"WMI MOF class", L"WPP
TMF file"};

// Handle to the trace file that you opened.

TRACEHANDLE g_hTrace = 0;

// Prototypes

void WINAPI ProcessEvent(PEVENT_RECORD pEvent);
DWORD GetEventInformation(PEVENT_RECORD pEvent, PTRACE_EVENT_INFO & pInfo);
DWORD PrintPropertyMetadata(TRACE_EVENT_INFO* pInfo, DWORD i, USHORT
indent);

void wmain(void)
{
    ULONG status = ERROR_SUCCESS;
```

```

EVENT_TRACE_LOGFILE trace;
TRACE_LOGFILE_HEADER* pHHeader = &trace.LogfileHeader;

// Identify the log file from which you want to consume events
// and the callbacks used to process the events and buffers.

ZeroMemory(&trace, sizeof(EVENT_TRACE_LOGFILE));
trace.LogFileName = (LPWSTR) LOGFILE_PATH;
trace.EventRecordCallback = (PEVENT_RECORD_CALLBACK) (ProcessEvent);
trace.ProcessTraceMode = PROCESS_TRACE_MODE_EVENT_RECORD;

g_hTrace = OpenTrace(&trace);
if (INVALID_PROCESSTRACE_HANDLE == g_hTrace)
{
    wprintf(L"OpenTrace failed with %lu\n", GetLastError());
    goto cleanup;
}

status = ProcessTrace(&g_hTrace, 1, 0, 0);
if (status != ERROR_SUCCESS && status != ERROR_CANCELLED)
{
    wprintf(L"ProcessTrace failed with %lu\n", status);
    goto cleanup;
}

cleanup:

if (INVALID_PROCESSTRACE_HANDLE != g_hTrace)
{
    status = CloseTrace(g_hTrace);
}
}

VOID WINAPI ProcessEvent(PEVENT_RECORD pEvent)
{
    DWORD status = ERROR_SUCCESS;
    HRESULT hr = S_OK;
    PTRACE_EVENT_INFO pInfo = NULL;
    LPWSTR pStringGuid = NULL;

    // Skips the event if it is the event trace header. Log files contain
    // this event
    // but real-time sessions do not. The event contains the same
    // information as
    // the EVENT_TRACE_LOGFILE.LogfileHeader member that you can access when
    // you open
    // the trace.

    if (IsEqualGUID(pEvent->EventHeader.ProviderId, EventTraceGuid) &&
        pEvent->EventHeader.EventDescriptor.Opcode == EVENT_TRACE_TYPE_INFO)
    {
        ; // Skip this event.
    }
}

```

```
else
{
    // Process the event. This example does not process the event data
but
    // instead prints the metadata that describes each event.

    status = GetEventInformation(pEvent, pInfo);

    if (ERROR_SUCCESS != status)
    {
        wprintf(L"GetEventInformation failed with %lu\n", status);
        goto cleanup;
    }

    wprintf(L"Decoding source: %s\n", pSource[pInfo->DecodingSource]);

    if (DecodingSourceWPP == pInfo->DecodingSource)
    {
        // This example is not rendering WPP metadata.
        goto cleanup;
    }

    if (pInfo->ProviderNameOffset > 0)
    {
        wprintf(L"Provider name: %s\n", (LPWSTR)((PBYTE)(pInfo) + pInfo-
>ProviderNameOffset));
    }

    hr = StringFromCLSID(pInfo->ProviderGuid, &pStringGuid);
    if (FAILED(hr))
    {
        wprintf(L"StringFromCLSID(ProviderGuid) failed with 0x%x\n",
hr);
        status = hr;
        goto cleanup;
    }

    wprintf(L"\nProvider GUID: %s\n", pStringGuid);
    CoTaskMemFree(pStringGuid);
    pStringGuid = NULL;

    if (!IsEqualGUID(pInfo->EventGuid, GUID_NULL))
    {
        hr = StringFromCLSID(pInfo->EventGuid, &pStringGuid);
        if (FAILED(hr))
        {
            wprintf(L"StringFromCLSID(EventGuid) failed with 0x%x\n",
hr);
            status = hr;
            goto cleanup;
        }

        wprintf(L"\nEvent GUID: %s\n", pStringGuid);
        CoTaskMemFree(pStringGuid);
        pStringGuid = NULL;
    }
}
```

```
}

    if (DecodingSourceXMLFile == pInfo->DecodingSource)
    {
        wprintf(L"Event ID: %hu\n", pInfo->EventDescriptor.Id);
    }

    wprintf(L"Version: %d\n", pInfo->EventDescriptor.Version);

    if (pInfo->ChannelNameOffset > 0)
    {
        wprintf(L"Channel name: %s\n", (LPWSTR)((PBYTE)(pInfo) + pInfo-
>ChannelNameOffset));
    }

    if (pInfo->LevelNameOffset > 0)
    {
        wprintf(L"Level name: %s\n", (LPWSTR)((PBYTE)(pInfo) + pInfo-
>LevelNameOffset));
    }
    else
    {
        wprintf(L"Level: %hu\n", pInfo->EventDescriptor.Level);
    }

    if (DecodingSourceXMLFile == pInfo->DecodingSource)
    {
        if (pInfo->OpcodeNameOffset > 0)
        {
            wprintf(L"Opcode name: %s\n", (LPWSTR)((PBYTE)(pInfo) +
pInfo->OpcodeNameOffset));
        }
    }
    else
    {
        wprintf(L"Type: %hu\n", pInfo->EventDescriptor.Opcde);
    }

    if (DecodingSourceXMLFile == pInfo->DecodingSource)
    {
        if (pInfo->TaskNameOffset > 0)
        {
            wprintf(L"Task name: %s\n", (LPWSTR)((PBYTE)(pInfo) + pInfo-
>TaskNameOffset));
        }
    }
    else
    {
        wprintf(L"Task: %hu\n", pInfo->EventDescriptor.Task);
    }

    wprintf(L"Keyword mask: 0x%08x\n", pInfo->EventDescriptor.Keyword);
    if (pInfo->KeywordsNameOffset)
    {
```

```

        LPWSTR pKeyword = (LPWSTR)((PBYTE)(pInfo) + pInfo-
>KeywordsNameOffset);

        for (; *pKeyword != 0; pKeyword += (wcslen(pKeyword) + 1))
            wprintf(L" Keyword name: %s\n", pKeyword);
    }

    if (pInfo->EventMessageOffset > 0)
    {
        wprintf(L"Event message: %s\n", (LPWSTR)((PBYTE)(pInfo) + pInfo-
>EventMessageOffset));
    }

    if (pInfo->ActivityIDNameOffset > 0)
    {
        wprintf(L"Activity ID name: %s\n", (LPWSTR)((PBYTE)(pInfo) +
pInfo->ActivityIDNameOffset));
    }

    if (pInfo->RelatedActivityIDNameOffset > 0)
    {
        wprintf(L"Related activity ID name: %s\n", (LPWSTR)((PBYTE)
(pInfo) + pInfo->RelatedActivityIDNameOffset));
    }

    wprintf(L"Number of top-level properties: %lu\n", pInfo-
>TopLevelPropertyCount);

    wprintf(L"Total number of properties: %lu\n", pInfo->PropertyCount);

    // Print the metadata for all the top-level properties. Metadata for
all the
    // top-level properties come before structure member properties in
the
    // property information array.

    if (pInfo->TopLevelPropertyCount > 0)
    {
        wprintf(L"\nThe following are the user data properties defined
for this event:\n");

        for (USHORT i = 0; i < pInfo->TopLevelPropertyCount; i++)
        {
            status = PrintPropertyMetadata(pInfo, i, 0);
            if (ERROR_SUCCESS != status)
            {
                wprintf(L"Printing metadata for top-level properties
failed.\n");
                goto cleanup;
            }
        }
    }
    else
    {
        wprintf(L"\nThe event does not define any user data

```

```

properties.\n");
}

wprintf(L"\n");
}

cleanup:

if (pInfo)
{
    free(pInfo);
}

if (ERROR_SUCCESS != status)
{
    CloseTrace(g_hTrace);
}
}

DWORD GetEventInformation(PEVENT_RECORD pEvent, PTRACE_EVENT_INFO & pInfo)
{
    DWORD status = ERROR_SUCCESS;
    DWORD BufferSize = 0;

    // Retrieve the required buffer size for the event metadata.

    status = TdhGetEventInformation(pEvent, 0, NULL, pInfo, &BufferSize);

    if (ERROR_INSUFFICIENT_BUFFER == status)
    {
        pInfo = (TRACE_EVENT_INFO*) malloc(BufferSize);
        if (pInfo == NULL)
        {
            wprintf(L"Failed to allocate memory for event info
(size=%lu).\n", BufferSize);
            status = ERROR_OUTOFMEMORY;
            goto cleanup;
        }

        // Retrieve the event metadata.

        status = TdhGetEventInformation(pEvent, 0, NULL, pInfo,
&BufferSize);
    }

    if (ERROR_SUCCESS != status)
    {
        wprintf(L"TdhGetEventInformation failed with 0x%x.\n", status);
    }
}

cleanup:

return status;

```

```

}

// Print the metadata for each property.

DWORD PrintPropertyMetadata(TRACE_EVENT_INFO* pinfo, DWORD i, USHORT indent)
{
    DWORD status = ERROR_SUCCESS;
    DWORD j = 0;
    DWORD lastMember = 0; // Last member of a structure

    // Print property name.

    wprintf(L"%*s%s", indent, L"", (LPWSTR)((PBYTE)(pinfo) + pinfo-
>EventPropertyInfoArray[i].NameOffset));

    // If the property is an array, the property can define the array size
    // or it can
    // point to another property whose value defines the array size. The
    PropertyParamCount
    // flag tells you where the array size is defined.

    if ((pinfo->EventPropertyInfoArray[i].Flags & PropertyParamCount) ==
    PropertyParamCount)
    {
        j = pinfo->EventPropertyInfoArray[i].countPropertyIndex;
        wprintf(L" (array size is defined by %s)", (LPWSTR)((PBYTE)(pinfo) +
    pinfo->EventPropertyInfoArray[j].NameOffset));
    }
    else
    {
        if (pinfo->EventPropertyInfoArray[i].count > 1)
            wprintf(L" (array size is %lu)", pinfo-
>EventPropertyInfoArray[i].count);
    }

    // If the property is a buffer, the property can define the buffer size
    // or it can
    // point to another property whose value defines the buffer size. The
    PropertyParamLength
    // flag tells you where the buffer size is defined.

    if ((pinfo->EventPropertyInfoArray[i].Flags & PropertyParamLength) ==
    PropertyParamLength)
    {
        j = pinfo->EventPropertyInfoArray[i].lengthPropertyIndex;
        wprintf(L" (size is defined by %s)", (LPWSTR)((PBYTE)(pinfo) +
    pinfo->EventPropertyInfoArray[j].NameOffset));
    }
    else
    {
        // Variable length properties such as structures and some strings do
        not have
    }
}

```

```

// length definitions.

    if (pinfo->EventPropertyInfoArray[i].length > 0)
        wprintf(L" (size is %lu bytes)", pinfo-
>EventPropertyInfoArray[i].length);
    else
        wprintf(L" (size  is unknown)");
}

wprintf(L"\n");

// If the property is a structure, print the members of the structure.

if ((pinfo->EventPropertyInfoArray[i].Flags & PropertyStruct) ==
PropertyStruct)
{
    wprintf(L"%*s(The property is a structure and has the following %hu
members):\n", 4, L"",
           pinfo->EventPropertyInfoArray[i].structType.NumOfStructMembers);

    lastMember = pinfo-
>EventPropertyInfoArray[i].structType.StructStartIndex +
               pinfo->EventPropertyInfoArray[i].structType.NumOfStructMembers;

    for (j = pinfo-
>EventPropertyInfoArray[i].structType.StructStartIndex; j < lastMember; j++)
    {
        PrintPropertyMetadata(pinfo, j, 4);
    }
}
else
{
    // You can use InType to determine the data type of the member and
OutType
    // to determine the output format of the data.

    if (pinfo->EventPropertyInfoArray[i].nonStructType.MapNameOffset)
    {
        // You can pass the name to the TdhGetEventMapInformation
function to
        // retrieve metadata about the value map.

        wprintf(L"%*s(Map attribute name is %s)\n", indent, L"",
               (PWCHAR)((PBYTE)(pinfo) + pinfo-
>EventPropertyInfoArray[i].nonStructType.MapNameOffset));
    }
}

return status;
}

```



# Retrieving Event Data Using MOF

Article • 07/16/2021 • 20 minutes to read

To consume event specific data, the consumer must know the format of the event data. If the provider used MOF to publish the format of the event data, you can use the MOF class to parse the event data. All the kernel events use MOF to publish the format of the event data. For information on publishing events, see [Publishing Your Event Schema](#).

Parsing the event data requires using the Windows Management Infrastructure (WMI) API. The ETW namespace where providers publish their MOF class is root\wmi. The ETW namespace contains three types of MOF classes: the provider MOF class, the event MOF class, and the event type MOF class. The event MOF class logically groups one or more event type MOF classes. The event type MOF class defines the actual event data.

An event MOF class contains a **Guid** class qualifier whose value must match the value in the **Header.Guid** member of the [\*\*EVENT\\_TRACE\*\*](#) structure. To ensure you have the correct version of the class, also compare the **EventVersion** class qualifier to the **Header.Class.Version** member of the [\*\*EVENT\\_TRACE\*\*](#) structure.

After you find the correct event class, enumerate its child event type classes to find the class that contains the format of the event data. The correct event type class contains an **EventType** class qualifier whose value matches the value in the **Header.Class.Type** member of the [\*\*EVENT\\_TRACE\*\*](#) structure.

You can then use the WMI API to enumerate the properties of the MOF class. Use the qualifiers and data type of each property to determine the size of the data element in the event data to read and how to format it. For a list of MOF qualifiers that ETW supports, see [Event Tracing MOF Qualifiers](#).

Because ETW does not force an alignment between event data values, typecasting or assigning the value directly from a buffer may cause an alignment fault; you should not create a structure from the MOF class and try to use it to consume event data. For example, if you have a character followed by ULONGLONG, the ULONGLONG would not be aligned to an 8-byte boundary, so an assignment would cause an alignment exception. (On 64-bit computers, this happens more often.) For this reason, you should use CopyMemory to copy the data from the buffer to a local variable. Also, if the event is later revised, your consumer may not work if you try to use a structure.

Beginning with Windows Vista, you are encouraged to use the trace data helper (TDH) functions to consume events that were published using MOF classes. For details, see [Retrieving Event Data Using TDH](#).

The following example shows how to consume events that are defined by a MOF class.

C++

```
//Turns the DEFINE_GUID for EventTraceGuid into a const.
#define INITGUID

#include <windows.h>
#include <stdio.h>
#include <comutil.h>
#include <wbemidl.h>
#include <wmistr.h>
#include <evntrace.h>
#include <in6addr.h>

#pragma comment(lib, "comsupp.lib") // For _bstr_t class
#pragma comment(lib, "ws2_32.lib") // For ntohs function

#define LOGFILE_PATH L"<FULLPATHTOLOGFILE.etl>"

// Macro for determining the length of a SID.
#define SeLengthSid( Sid ) \
(8 + (4 * ((SID*)Sid)->SubAuthorityCount))

typedef struct _propertyList
{
    BSTR Name;      // Property name
    LONG CimType;   // Property data type
    IWbemQualifierSet* pQualifiers;
} PROPERTY_LIST;

// Used to determine the data size of property values that contain the
// Pointer or SizeT qualifier. The value will be 4 or 8.
USHORT g_PointerSize = 0;

// Used to calculate CPU usage
ULONG g_TimerResolution = 0;

// Used to determine if the session is a private session or kernel session.
// You need to know this when accessing some members of the
EVENT_TRACE.Header
// member (for example, KernelTime or UserTime).
BOOL g_bUserMode = FALSE;

// Used to terminate event processing early
BOOL g_TerminateProcessing = TRUE;

// Points to WMI namespace that contains the ETW MOF classes.
IWbemServices* g_pServices = NULL;

void WINAPI ProcessEvent(PEVENT_TRACE pEvent);
ULONG WINAPI ProcessBuffer(PEVENT_TRACE_LOGFILE pBuffer);
HRESULT ConnectToETWNamespace(BSTR bstrNamespace);
```

```

IWbemClassObject* GetEventCategoryClass(BSTR bstrClassGuid, ULONG Version);
IWbemClassObject* GetEventClass(IWbemClassObject* pEventTraceClass, ULONG
EventType);
BOOL GetPropertyList(IWbemClassObject* pClass, PROPERTY_LIST** ppProperties,
DWORD* pPropertyCount, LONG** ppPropertyIndex);
void FreePropertyList(PROPERTY_LIST* pProperties, DWORD Count, LONG*
pIndex);
void PrintPropertyName(PROPERTY_LIST* pProperty);
PBYTE PrintEventPropertyValue(PROPERTY_LIST* pProperty, PBYTE pEventData,
USHORT RemainingBytes);

typedef LPTSTR (NTAPI *PIPV6ADDRTOSTRING)(
    const IN6_ADDR *Addr,
    LPTSTR S
);

void wmain(void)
{
    ULONG status = ERROR_SUCCESS;
    EVENT_TRACE_LOGFILE trace;
    TRACE_LOGFILE_HEADER* pHeader = &trace.LogfileHeader;
    TRACEHANDLE hTrace = 0;
    HRESULT hr = S_OK;

    // Identify the log file from which you want to consume events
    // and the callbacks used to process the events and buffers.

    ZeroMemory(&trace, sizeof(EVENT_TRACE_LOGFILE));
    trace.LogFileName = (LPWSTR) LOGFILE_PATH;
    trace.BufferCallback = (PEVENT_TRACE_BUFFER_CALLBACK) (ProcessBuffer);
    trace.EventCallback = (PEVENT_CALLBACK) (ProcessEvent);

    hTrace = OpenTrace(&trace);
    if ((TRACEHANDLE)INVALID_HANDLE_VALUE == hTrace)
    {
        wprintf(L"OpenTrace failed with %lu\n", GetLastError());
        goto cleanup;
    }

    g_PointerSize = (USHORT)pHeader->PointerSize;
    g_bUserMode = pHeader->LogFileMode & EVENT_TRACE_PRIVATE_LOGGER_MODE;

    if (pHeader->TimerResolution > 0)
    {
        g_TimerResolution = pHeader->TimerResolution / 10000;
    }

    wprintf(L"Number of events lost: %lu\n", pHeader->EventsLost);

    // Use pHeader to access all fields prior to LoggerName.
    // Adjust pHeader based on the pointer size to access
    // all fields after LogFileName. This is required only if
    // you are consuming events on an architecture that is
    // different from architecture used to write the events.
}

```

```

if (pHeader->PointerSize != sizeof(PVOID))
{
    pHeader = (PTRACE_LOGFILE_HEADER)((PUCHAR)pHeader +
        2 * (pHeader->PointerSize - sizeof(PVOID)));
}

wprintf(L"Number of buffers lost: %lu\n\n", pHeader->BuffersLost);

hr = ConnectToETWNamespace(_bstr_t(L"root\\wmi"));
if (FAILED(hr))
{
    wprintf(L"ConnectToETWNamespace failed with 0x%x\n", hr);
    goto cleanup;
}

status = ProcessTrace(&hTrace, 1, 0, 0);
if (status != ERROR_SUCCESS && status != ERROR_CANCELLED)
{
    wprintf(L"ProcessTrace failed with %lu\n", status);
    goto cleanup;
}

cleanup:

if ((TRACEHANDLE)INVALID_HANDLE_VALUE != hTrace)
{
    status = CloseTrace(hTrace);
}

if (g_pServices)
{
    g_pServices->Release();
    g_pServices = NULL;
}

CoUninitialize();
}

VOID WINAPI ProcessEvent(PEVENT_TRACE pEvent)
{
    WCHAR ClassGuid[50];
    IWbemClassObject* pEventCategoryClass = NULL;
    IWbemClassObject* pEventClass = NULL;
    PBYTE pEventData = NULL;
    PBYTE pEndOfEventData = NULL;
    PROPERTY_LIST* pProperties = NULL;
    DWORD PropertyCount = 0;
    LONG* pPropertyIndex = NULL;
    ULONGLONG TimeStamp = 0;
    ULONGLONG Nanoseconds = 0;
    SYSTEMTIME st;
    SYSTEMTIME stLocal;
    FILETIME ft;
}

```

```

    // Skips the event if it is the event trace header. Log files contain
    this event
    // but real-time sessions do not. The event contains the same
    information as
    // the EVENT_TRACE_LOGFILE.LogfileHeader member that you can access when
    you open
    // the trace.

    if (IsEqualGUID(pEvent->Header.Guid, EventTraceGuid) &&
        pEvent->Header.Class.Type == EVENT_TRACE_TYPE_INFO)
    {
        ; // Skip this event.
    }
    else
    {
        // Process the event. The pEvent->MofData member is a pointer to
        // the event specific data, if it exists.

        // If you encounter an error while processing an event, you could
        // set g_TerminateProcessing to TRUE to terminate event processing
        // (will occur the next time ProcessBuffer is called).

        // Get the class GUID from the header and convert it into a string.

        StringFromGUID2(pEvent->Header.Guid, ClassGuid, sizeof(ClassGuid));
        wprintf(L"%s\n", ClassGuid);
        wprintf(L"EventVersion(%d)\n", pEvent->Header.Class.Version);
        wprintf(L"EventType(%d)\n", pEvent->Header.Class.Type);

        // Print the time stamp for when the event occurred.

        ft.dwHighDateTime = pEvent->Header.TimeStamp.HighPart;
        ft.dwLowDateTime = pEvent->Header.TimeStamp.LowPart;

        FileTimeToSystemTime(&ft, &st);
        SystemTimeToTzSpecificLocalTime(NULL, &st, &stLocal);

        TimeStamp = pEvent->Header.TimeStamp.QuadPart;
        Nanoseconds = (TimeStamp % 10000000) * 100;

        wprintf(L"%02d/%02d/%02d %02d:%02d:%02d.%I64u\n",
               stLocal.wMonth, stLocal.wDay, stLocal.wYear, stLocal.wHour,
               stLocal.wMinute, stLocal.wSecond, Nanoseconds);

        // If the event contains event-specific data find the MOF class that
        // contains the format of the event data.

        if (pEvent->MofLength > 0)
        {
            pEventCategoryClass = GetEventCategoryClass(_bstr_t(ClassGuid),
pEvent->Header.Class.Version);
            if (pEventCategoryClass)
            {

```

```

// Get the event class that contains the format of the event
data.

    pEventClass = GetEventClass(pEventCategoryClass, pEvent-
>Header.Class.Type);

    pEventCategoryClass->Release();
    pEventCategoryClass = NULL;

    if (pEventClass)
    {
        // Enumerate the properties and retrieve the event data.

        if (TRUE == GetPropertyList(pEventClass, &pProperties,
&PropertyCount, &pPropertyIndex))
        {
            // Print the property name and value.

            // Get a pointer to the beginning and end of the
            event data.
            // These pointers are used to calculate the number
            of bytes of event
            // data left to read. This is only useful if the
            last data
            // element is a string that contains the
            StringTermination("NotCounted") qualifier.

            pEventData = (PBYTE)(pEvent->MofData);
            pEndOfEventData = ((PBYTE)(pEvent->MofData) +
pEvent->MofLength);

            for (LONG i = 0; (DWORD)i < PropertyCount; i++)
            {

PrintPropertyName(pProperties+pPropertyIndex[i]);

                pEventData =
PrintEventPropertyValue(pProperties+pPropertyIndex[i],
                                         pEventData,
                                         (USHORT)
(pEndOfEventData - pEventData));

                if (NULL == pEventData)
                {
                    //Error reading the data. Handle as
                    appropriate for your application.
                    break;
                }
            }

            FreePropertyList(pProperties, PropertyCount,
pPropertyIndex);
        }

        pEventClass->Release();
    }
}

```

```

                pEventClass = NULL;
            }
            else
            {
                wprintf(L"Unable to find the MOF class for the
event.\n");
            }
        }
        else
        {
            wprintf(L"Unable to find MOF class for %s and version
%d.\n", ClassGuid, pEvent->Header.Class.Version);
        }

        wprintf(L"\n");
    }
}
}

ULONG WINAPI ProcessBuffer(PEVENT_TRACE_LOGFILE pBuffer)
{
    UNREFERENCED_PARAMETER(pBuffer);

    return g_TerminateProcessing;
}

HRESULT ConnectToETWNamespace(BSTR bstrNamespace)
{
    HRESULT hr = S_OK;
    IwbemLocator* pLocator = NULL;

    hr = CoInitialize(0);

    hr = CoCreateInstance(__uuidof(IwbemLocator),
        0,
        CLSCTX_INPROC_SERVER,
        __uuidof(IwbemLocator),
        (LPVOID*) &pLocator);

    if (FAILED(hr))
    {
        wprintf(L"CoCreateInstance failed with 0x%x\n", hr);
        goto cleanup;
    }

    hr = pLocator->ConnectServer(bstrNamespace,
        NULL, NULL, NULL,
        0L, NULL, NULL,
        &g_pServices);

    if (FAILED(hr))
    {
        wprintf(L"pLocator->ConnectServer failed with 0x%x\n", hr);
    }
}
```

```

        goto cleanup;
    }

    hr = CoSetProxyBlanket(g_pServices,
        RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE,
        NULL,
        RPC_C_AUTHN_LEVEL_PKT, RPC_C_IMP_LEVEL_IMPERSONATE,
        NULL, EOAC_NONE);

    if (FAILED(hr))
    {
        wprintf(L"CoSetProxyBlanket failed with 0x%x\n", hr);
        g_pServices->Release();
        g_pServices = NULL;
    }

cleanup:

    if (pLocator)
        pLocator->Release();

    return hr;
}

IWbemClassObject* GetEventCategoryClass(BSTR bstrClassGuid, int Version)
{
    HRESULT hr = S_OK;
    HRESULT hrQualifier = S_OK;
    IEnumWbemClassObject* pClasses = NULL;
    IWbemClassObject* pClass = NULL;
    IWbemQualifierSet* pQualifiers = NULL;
    ULONG cnt = 0;
    VARIANT varGuid;
    VARIANT varVersion;

    // All ETW MOF classes derive from the EventTrace class, so you need to
    // enumerate all the EventTrace descendants to find the correct event
    // category class.

    hr = g_pServices->CreateClassEnum(_bstr_t(L"EventTrace"),
        WBEM_FLAG_DEEP | WBEM_FLAG_FORWARD_ONLY |
        WBEM_FLAG_USE_AMENDED_QUALIFIERS,
        NULL, &pClasses);

    if (FAILED(hr))
    {
        wprintf(L"g_pServices->CreateClassEnum failed with 0x%x\n", hr);
        goto cleanup;
    }

    while (S_OK == hr)
    {
        hr = pClasses->Next(WBEM_INFINITE, 1, &pClass, &cnt);

```

```

    if (FAILED(hr))
    {
        wprintf(L"pClasses->Next failed with 0x%x\n", hr);
        goto cleanup;
    }

    // Get all the qualifiers for the class and search for the Guid
    // qualifier.

    hrQualifier = pClass->GetQualifierSet(&pQualifiers);

    if (pQualifiers)
    {
        hrQualifier = pQualifiers->Get(L"Guid", 0, &varGuid, NULL);

        if (SUCCEEDED(hrQualifier))
        {
            // Compare this class' GUID to the one from the event.

            if (_wcsicmp(varGuid.bstrVal, bstrClassGuid) == 0)
            {
                // If the GUIDs are equal, check for the correct
                // version.
                if (EventVersion < 0)
                {
                    // The version is correct if the class does not contain
                    // the EventVersion
                    // qualifier or the class version matches the version
                    // from the event.
                }
            }

            hrQualifier = pQualifiers->Get(L"EventVersion", 0,
                &varVersion, NULL);

            if (SUCCEEDED(hrQualifier))
            {
                if (Version == varVersion.intValue)
                {
                    break; //found class
                }

                VariantClear(&varVersion);
            }
            else if (WBEM_E_NOT_FOUND == hrQualifier)
            {
                break; //found class
            }
        }
    }

    VariantClear(&varGuid);
}

pQualifiers->Release();
pQualifiers = NULL;
}

pClass->Release();

```

```

        pClass = NULL;
    }

cleanup:

    if (pClasses)
    {
        pClasses->Release();
        pClasses = NULL;
    }

    if (pQualifiers)
    {
        pQualifiers->Release();
        pQualifiers = NULL;
    }

    VariantClear(&varVersion);
    VariantClear(&varGuid);

    return pClass;
}

IWbemClassObject* GetEventClass(IWbemClassObject* pEventCategoryClass, int
EventType)
{
    HRESULT hr = S_OK;
    HRESULT hrQualifier = S_OK;
    IEnumWbemClassObject* pClasses = NULL;
    IWbemClassObject* pClass = NULL;
    IWbemQualifierSet* pQualifiers = NULL;
    ULONG cnt = 0;
    VARIANT varClassName;
    VARIANT varEventType;
    BOOL FoundEventClass = FALSE;

    // Get the name of the event category class so you can enumerate its
children classes.

    hr = pEventCategoryClass->Get(L"__RELPATH", 0, &varClassName, NULL,
NULL);

    if (FAILED(hr))
    {
        wprintf(L"pEventCategoryClass->Get failed with 0x%x\n", hr);
        goto cleanup;
    }

    hr = g_pServices->CreateClassEnum(varClassName.bstrVal,
        WBEM_FLAG_SHALLOW | WBEM_FLAG_FORWARD_ONLY |
WBEM_FLAG_USE_AMENDED_QUALIFIERS,
        NULL, &pClasses);

    if (FAILED(hr))

```

```

    {
        wprintf(L"g_pServices->CreateClassEnum failed with 0x%x\n", hr);
        goto cleanup;
    }

    // Loop through the enumerated classes and find the event class that
    // matches the event. The class is a match if the event type from the
    // event matches the value from the EventType class qualifier.

    while (S_OK == hr)
    {
        hr = pClasses->Next(WBEM_INFINITE, 1, &pClass, &cnt);

        if (FAILED(hr))
        {
            wprintf(L"pClasses->Next failed with 0x%x\n", hr);
            goto cleanup;
        }

        // Get all the qualifiers for the class and search for the EventType
        // qualifier.

        hrQualifier = pClass->GetQualifierSet(&pQualifiers);

        if (FAILED(hrQualifier))
        {
            wprintf(L"pClass->GetQualifierSet failed with 0x%x\n",
hrQualifier);
            pClass->Release();
            pClass = NULL;
            goto cleanup;
        }

        hrQualifier = pQualifiers->Get(L"EventType", 0, &varEventType,
NULL);

        if (FAILED(hrQualifier))
        {
            wprintf(L"pQualifiers->Get(eventtype) failed with 0x%x\n",
hrQualifier);
            pClass->Release();
            pClass = NULL;
            goto cleanup;
        }

        // If multiple events provide the same data, the EventType qualifier
        // will contain an array of types. Loop through the array and find a
        // match.

        if (varEventType.vt & VT_ARRAY)
        {
            HRESULT hrSafe = S_OK;
            int ClassEventType;
            SAFEARRAY* pEventTypes = varEventType.parray;

```

```

        for (LONG i=0; (ULONG)i < pEventTypes->rgsabound->cElements;
i++)
{
    hrSafe = SafeArrayGetElement(pEventTypes, &i,
&ClassEventType);

    if (ClassEventType == EventType)
    {
        FoundEventClass = TRUE;
        break; //for loop
    }
}
else
{
    if (varEventType.intVal == EventType)
    {
        FoundEventClass = TRUE;
    }
}

VariantClear(&varEventType);

if (TRUE == FoundEventClass)
{
    break; //while loop
}

pClass->Release();
pClass = NULL;
}

cleanup:

if (pClasses)
{
    pClasses->Release();
    pClasses = NULL;
}

if (pQualifiers)
{
    pQualifiers->Release();
    pQualifiers = NULL;
}

VariantClear(&varClassName);
VariantClear(&varEventType);

return pClass;
}

// This function retrieves the list of properties, data type, and qualifiers
for

```

```

// each property in the class. If you know the name of the property you want
to
// retrieve, you can call the IWbemClassObject::Get method to retrieve the
data
// type and IWbemClassObject::GetPropertyQualifierSet to retrieve its
qualifiers.

BOOL GetPropertyList(IWbemClassObject* pClass, PROPERTY_LIST** ppProperties,
DWORD* pPropertyCount, LONG** ppPropertyIndex)
{
    HRESULT hr = S_OK;
    SAFEARRAY* pNames = NULL;
    LONG j = 0;
    VARIANT var;

    // Retrieve the property names.

    hr = pClass->GetNames(NULL, WBEM_FLAG_NONSYSTEM_ONLY, NULL, &pNames);
    if (pNames)
    {
        *pPropertyCount = pNames->rgsabound->cElements;

        // Allocate a block of memory to hold an array of PROPERTY_LIST
structures.

        *ppProperties = (PROPERTY_LIST*) malloc(sizeof(PROPERTY_LIST) *
(*pPropertyCount));
        if (NULL == *ppProperties)
        {
            hr = E_OUTOFMEMORY;
            goto cleanup;
        }

        // WMI may not return the properties in the order as defined in the
MOF. Allocate
        // an array of indexes that map into the property list array, so you
can retrieve
        // the event data in the correct order.

        *ppPropertyIndex = (LONG*) malloc(sizeof(LONG) * (*pPropertyCount));
        if (NULL == *ppPropertyIndex)
        {
            hr = E_OUTOFMEMORY;
            goto cleanup;
        }

        for (LONG i = 0; (ULONG)i < *pPropertyCount; i++)
        {
            //Save the name of the property.

            hr = SafeArrayGetElement(pNames, &i, &(*ppProperties+i)->Name));
            if (FAILED(hr))
            {
                goto cleanup;
            }
        }
    }
}

```

```

    }

        //Save the qualifiers. Used latter to help determine how to read
        //the event data.

        hr = pClass->GetPropertyQualifierSet((*ppProperties+i)->Name, &
        ((*ppProperties+i)->pQualifiers));
        if (FAILED(hr))
        {
            goto cleanup;
        }

        // Use the WmiDataId qualifier to determine the correct property
        // order.
        // Index[0] points to the property list element that contains
        WmiDataId("1"),
        // Index[1] points to the property list element that contains
        WmiDataId("2"),
        // and so on.

        hr = (*ppProperties+i)->pQualifiers->Get(L"WmiDataId", 0, &var,
        NULL);
        if (SUCCEEDED(hr))
        {
            j = var.intVal - 1;
            VariantClear(&var);
            *(*ppPropertyIndex+j) = i;
        }
        else if (WBEM_E_NOT_FOUND == hr)
        {
            continue; // Ignore property without WmiDataId
        }
        else
        {
            goto cleanup;
        }

        // Save the data type of the property.

        hr = pClass->Get((*ppProperties+i)->Name, 0, NULL, &
        ((*ppProperties+i)->CimType), NULL);
        if (FAILED(hr))
        {
            goto cleanup;
        }
    }

cleanup:

    if (pNames)
    {
        SafeArrayDestroy(pNames);
    }

```

```

    if (FAILED(hr))
    {
        if (*ppProperties)
        {
            FreePropertyList(*ppProperties, *pPropertyCount,
*pPropertyIndex);
        }

        return FALSE;
    }

    return TRUE;
}

void FreePropertyList(PROPERTY_LIST* pProperties, DWORD Count, LONG* pIndex)
{
    if(pProperties)
    {
        for (DWORD i=0; i < Count; i++)
        {
            SysFreeString((pProperties+i)->Name);

            if ((pProperties+i)->pQualifiers)
            {
                (pProperties+i)->pQualifiers->Release();
                (pProperties+i)->pQualifiers = NULL;
            }
        }

        free(pProperties);
    }

    if (pIndex)
        free(pIndex);
}
}

void PrintPropertyName(PROPERTY_LIST* pProperty)
{
    HRESULT hr;
    VARIANT varDisplayName;

    // Retrieve the Description qualifier for the property. The description
    // qualifier
    // should contain a printable display name for the property. If the
    // qualifier is
    // not found, print the property name.

    hr = pProperty->pQualifiers->Get(L"Description", 0, &varDisplayName,
NULL);
    wprintf(L"%s: ", (SUCCEEDED(hr)) ? varDisplayName.bstrVal : pProperty-
>Name);
    VariantClear(&varDisplayName);
}

```

```

PBYTE PrintEventPropertyValue(PROPERTY_LIST* pProperty, PBYTE pEventData,
USHORT RemainingBytes)
{
    HRESULT hr;
    VARIANT varQualifier;
    ULONG ArraySize = 1;
    BOOL PrintAsChar = FALSE;
    BOOL PrintAsHex = FALSE;
    BOOL PrintAsIPAddress = FALSE;
    BOOL PrintAsPort = FALSE;
    BOOL IsWideString = FALSE;
    BOOL IsNullTerminated = FALSE;
    USHORT StringLength = 0;

    // If the property contains the Pointer or PointerType qualifier,
    // you do not need to know the data type of the property. You just
    // retrieve either four bytes or eight bytes depending on the
    // pointer's size.

    if (SUCCEEDED(hr = pProperty->pQualifiers->Get(L"Pointer", 0, NULL,
NULL)) ||
        SUCCEEDED(hr = pProperty->pQualifiers->Get(L"PointerType", 0, NULL,
NULL)))
    {
        if (g_PointerSize == 4)
        {
            ULONG temp = 0;

            CopyMemory(&temp, pEventData, sizeof(ULONG));
            wprintf(L"0x%llx\n", temp);
        }
        else
        {
            ULLONG temp = 0;

            CopyMemory(&temp, pEventData, sizeof(ULLONG));
            wprintf(L"0x%llx\n", temp);
        }

        pEventData += g_PointerSize;
    }
    return pEventData;
}
else
{
    // If the property is an array, retrieve its size. The ArraySize
variable
    // is initialized to 1 to force the loops below to print the value
    // of the property.

    if (pProperty->CimType & CIM_FLAG_ARRAY)
    {
        hr = pProperty->pQualifiers->Get(L"MAX", 0, &varQualifier,

```

```

NULL);
    if (SUCCEEDED(hr))
    {
        ArraySize = varQualifier.intVal;
        VariantClear(&varQualifier);
    }
    else
    {
        wprintf(L"Failed to retrieve the MAX qualifier.
Terminating.\n");
        return NULL;
    }
}

// The CimType is the data type of the property.

switch(pProperty->CimType & (~CIM_FLAG_ARRAY))
{
    case CIM_SINT32:
    {
        LONG temp = 0;

        for (ULONG i=0; i < ArraySize; i++)
        {
            CopyMemory(&temp, pEventData, sizeof(LONG));
            wprintf(L"%d\n", temp);
            pEventData += sizeof(LONG);
        }

        return pEventData;
    }

    case CIM_UINT32:
    {
        ULONG temp = 0;

        hr = pProperty->pQualifiers->Get(L"Extension", 0,
&varQualifier, NULL);
        if (SUCCEEDED(hr))
        {
            // Some kernel events pack an IP address into a UINT32.
            // Check for an Extension qualifier whose value is
IPAddr.
            // This is here to support legacy event classes; the
IPAddr extension
            // should only be used on properties whose CIM type is
object.

            if (_wcsicmp(L"IPAddr", varQualifier.bstrVal) == 0)
            {
                PrintAsIPAddress = TRUE;
            }

            VariantClear(&varQualifier);
        }
    }
}

```

```

        else
        {
            hr = pProperty->pQualifiers->Get(L"Format", 0, NULL,
NULL);
            if (SUCCEEDED(hr))
            {
                PrintAsHex = TRUE;
            }
        }

        for (ULONG i = 0; i < ArraySize; i++)
        {
            CopyMemory(&temp, pEventData, sizeof(ULONG));

            if (PrintAsIPAddress)
            {
                wprintf(L"%03d.%03d.%03d.%03d\n", (temp >> 0) &
0xff,
                                                (temp >> 8) &
0xff,
                                                (temp >> 16) &
0xff,
                                                (temp >> 24) &
0xff);
            }
            else if (PrintAsHex)
            {
                wprintf(L"0x%llx\n", temp);
            }
            else
            {
                wprintf(L"%lu\n", temp);
            }

            pEventData += sizeof(ULONG);
        }

        return pEventData;
    }

case CIM_SINT64:
{
    LONGLONG temp = 0;

    for (ULONG i=0; i < ArraySize; i++)
    {
        CopyMemory(&temp, pEventData, sizeof(LONGLONG));
        wprintf(L"%I64d\n", temp);
        pEventData += sizeof(LONGLONG);
    }

    return pEventData;
}

case CIM_UINT64:

```

```

    {

        ULONGLONG temp = 0;

        for (ULONG i=0; i < ArraySize; i++)
        {
            CopyMemory(&temp, pEventData, sizeof(ULLONG));
            wprintf(L"%I64u\n", temp);
            pEventData += sizeof(ULLONG);
        }

        return pEventData;
    }

    case CIM_STRING:
    {
        USHORT temp = 0;

        // The format qualifier is included only if the string is a
        // wide string.

        hr = pProperty->pQualifiers->Get(L"Format", 0, NULL, NULL);
        if (SUCCEEDED(hr))
        {
            IsWideString = TRUE;
        }

        hr = pProperty->pQualifiers->Get(L"StringTermination", 0,
        &varQualifier, NULL);
        if (FAILED(hr) || (_wcsicmp(varQualifier.bstrVal,
        L"NullTerminated") == 0))
        {
           IsNullTerminated = TRUE;
        }
        else if (_wcsicmp(varQualifier.bstrVal, L"Counted") == 0)
        {
            // First two bytes of the string contain its length.

            CopyMemory(&StringLength, pEventData, sizeof(USHORT));
            pEventData += sizeof(USHORT);
        }
        else if (_wcsicmp(varQualifier.bstrVal, L"ReverseCounted")
        == 0)
        {
            // First two bytes of the string contain its length.
            // Count is in big-endian; convert to little-endian.

            CopyMemory(&temp, pEventData, sizeof(USHORT));
            StringLength = MAKEWORD(HIBYTE(temp), LOBYTE(temp));
            pEventData += sizeof(USHORT);
        }
        else if (_wcsicmp(varQualifier.bstrVal, L"NotCounted") == 0)
        {
            // The string is not null-terminated and does not
            // contain
            // its own length, so its length is the remaining bytes
        }
    }
}

```

```

of
    // the event data.

    StringLength = RemainingBytes;
}

VariantClear(&varQualifier);

for (ULONG i = 0; i < ArraySize; i++)
{
    if (IsWideString)
    {
        if (IsNullTerminated)
        {
            StringLength =
(USHORT)wcslen((WCHAR*)pEventData) + 1;
            wprintf(L"%s\n", (WCHAR*)pEventData);
        }
        else
        {
            LONG StringSize = (StringLength) *
sizeof(WCHAR);
            WCHAR* pwsz = (WCHAR*)malloc(StringSize+2); // +2 for NULL

            if (pwsz)
            {
                CopyMemory(pwsz, (WCHAR*)pEventData,
StringSize);
                *(pwsz+StringSize) = '\0';
                wprintf(L"%s\n", pwsz);
                free(pwsz);
            }
            else
            {
                // Handle allocation error.
            }
        }
        StringLength *= sizeof(WCHAR);
    }
    else // It is an ANSI string
    {
        if (IsNullTerminated)
        {
            StringLength = (USHORT)strlen((char*)pEventData)
+ 1;
            printf("%s\n", (char*)pEventData);
        }
        else
        {
            char* psz = (char*)malloc(StringLength+1); // +1 for NULL

            if (psz)

```

```

        {
            CopyMemory(psz, (char*)pEventData,
StringLength);
                *(psz+StringLength) = '\0';
                printf("%s\n", psz);
                free(psz);
        }
    else
    {
        // Handle allocation error.
    }
}
}

pEventData += StringLength;
StringLength = 0;
}

return pEventData;
}

case CIM_BOOLEAN:
{
    BOOL temp = FALSE;

    for (ULONG i=0; i < ArraySize; i++)
    {
        CopyMemory(&temp, pEventData, sizeof(BOOL));
        wprintf(L"%s\n", (temp) ? L"TRUE" : L"FALSE");
        pEventData += sizeof(BOOL);
    }

    return pEventData;
}

case CIM_SINT8:
case CIM_UINT8:
{
    hr = pProperty->pQualifiers->Get(L"Extension", 0,
&varQualifier, NULL);
    if (SUCCEEDED(hr))
    {
        // This is here to support legacy event classes; the
Guid extension
        // should only be used on properties whose CIM type is
object.

        if (_wcsicmp(L"Guid", varQualifier.bstrVal) == 0)
        {
            WCHAR szGuid[50];
            GUID Guid;

            CopyMemory(&Guid, (GUID*)pEventData, sizeof(GUID));
            StringFromGUID2(Guid, szGuid, sizeof(szGuid)-1);
            wprintf(L"%s\n", szGuid);
        }
    }
}
}

```

```

        }

        VariantClear(&varQualifier);
        pEventData += sizeof(GUID);
    }
    else
    {
        hr = pProperty->pQualifiers->Get(L"Format", 0, NULL,
NULL);

        if (SUCCEEDED(hr))
        {
            PrintAsChar = TRUE; // ANSI character
        }

        for (ULONG i = 0; i < ArraySize; i++)
        {
            if (PrintAsChar)
                wprintf(L"%c", *((char*)pEventData));
            else
                wprintf(L"%hd", *((BYTE*)pEventData));

            pEventData += sizeof(UINT8);
        }
    }

    wprintf(L"\n");

    return pEventData;
}

case CIM_CHAR16:
{
    WCHAR temp;

    for (ULONG i = 0; i < ArraySize; i++)
    {
        CopyMemory(&temp, pEventData, sizeof(WCHAR));
        wprintf(L"%c", temp);
        pEventData += sizeof(WCHAR);
    }

    wprintf(L"\n");

    return pEventData;
}

case CIM_SINT16:
{
    SHORT temp = 0;

    for (ULONG i = 0; i < ArraySize; i++)
    {
        CopyMemory(&temp, pEventData, sizeof(SHORT));
        wprintf(L"%hd\n", temp);
        pEventData += sizeof(SHORT);
    }
}

```

```

        }

        return pEventData;
    }

    case CIM_UINT16:
    {
        USHORT temp = 0;

        // If the data is a port number, call the ntohs Windows
        // Socket 2 function
        // to convert the data from TCP/IP network byte order to
        // host byte order.
        // This is here to support legacy event classes; the Port
        // extension
        // should only be used on properties whose CIM type is
        // object.

        hr = pProperty->pQualifiers->Get(L"Extension", 0,
        &varQualifier, NULL);
        if (SUCCEEDED(hr))
        {
            if (_wcsicmp(L"Port", varQualifier.bstrVal) == 0)
            {
                PrintAsPort = TRUE;
            }

            VariantClear(&varQualifier);
        }

        for (ULONG i = 0; i < ArraySize; i++)
        {
            CopyMemory(&temp, pEventData, sizeof(USHORT));

            if (PrintAsPort)
            {
                wprintf(L"%hu\n", ntohs(temp));
            }
            else
            {
                wprintf(L"%hu\n", temp);
            }

            pEventData += sizeof(USHORT);
        }

        return pEventData;
    }

    case CIM_OBJECT:
    {
        // An object data type has to include the Extension
        // qualifier.

        hr = pProperty->pQualifiers->Get(L"Extension", 0,

```

```

&varQualifier, NULL);
    if (SUCCEEDED(hr))
    {
        if (_wcsicmp(L"SizeT", varQualifier.bstrVal) == 0)
        {
            VariantClear(&varQualifier);

                // You do not need to know the data type of the
                property, you just
                // retrieve either 4 bytes or 8 bytes depending on
                the pointer's size.

                    for (ULONG i = 0; i < ArraySize; i++)
                    {
                        if (g_PointerSize == 4)
                        {
                            ULONG temp = 0;

                                CopyMemory(&temp, pEventData,
sizeof(ULONG));
                                wprintf(L"0x%llx\n", temp);
                        }
                        else
                        {
                            ULONGLONG temp = 0;

                                CopyMemory(&temp, pEventData,
sizeof(ULLONG));
                                wprintf(L"0x%llx\n", temp);
                        }

                            pEventData += g_PointerSize;
                    }

                    return pEventData;
    }
    if (_wcsicmp(L"Port", varQualifier.bstrVal) == 0)
    {
        USHORT temp = 0;

        VariantClear(&varQualifier);

            for (ULONG i = 0; i < ArraySize; i++)
            {
                CopyMemory(&temp, pEventData, sizeof(USHORT));
                wprintf(L"%hu\n", ntohs(temp));
                pEventData += sizeof(USHORT);
            }

            return pEventData;
    }
    else if (_wcsicmp(L"IPAddr", varQualifier.bstrVal) ==
0)
|| _wcsicmp(L"IPAddrV4", varQualifier.bstrVal) ==
0)

```

```

    {
        ULONG temp = 0;

        VariantClear(&varQualifier);

        for (ULONG i = 0; i < ArraySize; i++)
        {
            CopyMemory(&temp, pEventData, sizeof(ULONG));

            wprintf(L"%d.%d.%d.%d\n", (temp >> 0) & 0xff,
                    (temp >> 8) & 0xff,
                    (temp >> 16) & 0xff,
                    (temp >> 24) & 0xff);

            pEventData += sizeof(ULONG);
        }

        return pEventData;
    }
    else if (_wcsicmp(L"IPAddrV6", varQualifier.bstrVal) ==
0)
{
    WCHAR IPv6AddressAsString[46];
    IN6_ADDR IPv6Address;
    PIPV6ADDRTOSTRING fnRtlIpv6AddressToString;

    VariantClear(&varQualifier);

    fnRtlIpv6AddressToString =
(PIPV6ADDRTOSTRING)GetProcAddress(
    GetModuleHandle(L"ntdll"),
    "RtlIpv6AddressToStringW");

    if (NULL == fnRtlIpv6AddressToString)
    {
        wprintf(L"GetProcAddress failed with %lu.\n",
GetLastError());
        return NULL;
    }

    for (ULONG i = 0; i < ArraySize; i++)
    {
        CopyMemory(&IPv6Address, pEventData,
sizeof(IN6_ADDR));

        fnRtlIpv6AddressToString(&IPv6Address,
IPv6AddressAsString);

        wprintf(L"%s\n", IPv6AddressAsString);

        pEventData += sizeof(IN6_ADDR);
    }

    return pEventData;
}

```

```

        else if (_wcsicmp(L"Guid", varQualifier.bstrVal) == 0)
        {
            WCHAR szGuid[50];
            GUID Guid;

            VariantClear(&varQualifier);

            for (ULONG i = 0; i < ArraySize; i++)
            {
                CopyMemory(&Guid, (GUID*)pEventData,
                           sizeof(GUID));

                StringFromGUID2(Guid, szGuid, sizeof(szGuid)-1);
                wprintf(L"%s\n", szGuid);

                pEventData += sizeof(GUID);
            }

            return pEventData;
        }
        else if (_wcsicmp(L"Sid", varQualifier.bstrVal) == 0)
        {
            // Get the user's security identifier and print the
            // user's name and domain.

            SID* psid;
            DWORD cchUserSize = 0;
            DWORD cchDomainSize = 0;
            WCHAR* pUser = NULL;
            WCHAR* pDomain = NULL;
            SID_NAME_USE eNameUse;
            DWORD status = 0;
            ULONG temp = 0;
            USHORT CopyLength = 0;
            BYTE buffer[SECURITY_MAX_SID_SIZE];

            VariantClear(&varQualifier);

            for (ULONG i = 0; i < ArraySize; i++)
            {
                CopyMemory(&temp, pEventData, sizeof(ULONG));

                if (temp > 0)
                {
                    // A property with the Sid extension is
                    // actually a
                    // TOKEN_USER structure followed by the SID.
                    // The size
                    // depending on
                    // 32-bit or
                    // is aligned
                    // of the TOKEN_USER structure differs
                    // whether the events were generated on a
                    // 64-bit architecture. Also the structure
                    // on an 8-byte boundary, so its size is 8
                }
            }
        }
    }
}

```

```
bytes on a
                                // 32-bit computer and 16 bytes on a 64-bit
computer.
                                // Doubling the pointer size handles both
cases.

        USHORT BytesToSid = g_PointerSize * 2;

        pEventData += BytesToSid;

        if (RemainingBytes - BytesToSid >
SECURITY_MAX_SID_SIZE)
{
    CopyLength = SECURITY_MAX_SID_SIZE;
}
else
{
    CopyLength = RemainingBytes -
BytesToSid;
}

CopyMemory(&buffer, pEventData, CopyLength);
psid = (SID*)&buffer;

        LookupAccountSid(NULL, psid, pUser,
&cchUserSize, pDomain, &cchDomainSize, &eNameUse);

        status = GetLastError();
        if (ERROR_INSUFFICIENT_BUFFER == status)
{
    pUser = (WCHAR*)malloc(cchUserSize *
sizeof(WCHAR));
    pDomain = (WCHAR*)malloc(cchDomainSize *
sizeof(WCHAR));

        if (pUser && pDomain)
{
            if (LookupAccountSid(NULL, psid,
pUser, &cchUserSize, pDomain, &cchDomainSize, &eNameUse))
{
                wprintf(L"%s\\%s\n", pDomain,
pUser);
            }
            else
{
                wprintf(L"Second
LookupAccountSid failed with, %d\n", GetLastError());
            }
        }
        else
{
            wprintf(L"Allocation error.\n");
        }
}

        if (pUser)
```

```

        {
            free(pUser);
            pUser = NULL;
        }

        if (pDomain)
        {
            free(pDomain);
            pDomain = NULL;
        }

        cchUserSize = 0;
        cchDomainSize = 0;
    }
    else if (ERROR_NONE_MAPPED == status)
    {
        wprintf(L"Unable to locate account for
the specified SID\n");
    }
    else
    {
        wprintf(L"First LookupAccountSid failed
with, %d\n", status);
    }

    pEventData += SeLengthSid(psid);
}
else // There is no SID
{
    pEventData += sizeof(ULONG);
}
}

return pEventData;
}
else
{
    wprintf(L"Extension, %s, not supported.\n",
varQualifier.bstrVal);
    VariantClear(&varQualifier);
    return NULL;
}
}
else
{
    wprintf(L"Object data type is missing Extension
qualifier.\n");
    return NULL;
}
}

default:
{
    wprintf(L"Unknown CIM type\n");
    return NULL;
}

```

```
    }

} // switch
}
```

# Using Events to Calculate CPU Usage

Article • 01/07/2021 • 3 minutes to read

The following example shows you how to use events to calculate the CPU usage for a set of instructions. This example assumes the provider wraps the instruction set with a start event type and an end event type.

C++

```
//Turns the DEFINE_GUID for EventTraceGuid into a const.
#define INITGUID

#include <windows.h>
#include <stdio.h>
#include <wbemidl.h>
#include <wmistr.h>
#include <evntrace.h>

#define LOGFILE_PATH L"<FULLPATHTOLOGFILE.etl>"

//Remember to use your own GUID. Event class GUID used in the provider.

// {12BF20F2-0B1C-47e8-90B3-13C81C7AFD9A}
static const GUID CpuUsageEvent =
{ 0x12bf20f2, 0xb1c, 0x47e8, { 0x90, 0xb3, 0x13, 0xc8, 0x1c, 0x7a, 0xfd,
0x9a } };

// Used to calculate CPU usage
ULONG g_TimerResolution = 0;

// Used to determine if the session is a private session or kernel session.
// You need to know this when accessing some members of the
EVENT_TRACE.Header
// member (for example, KernelTime or UserTime).
BOOL g_bUserMode = FALSE;

//Start time value for the start event.
ULONG g_StartKernelTime = 0;
ULONG g_StartUserTime = 0;
ULONG64 g_StartProcessTime = 0;

void WINAPI ProcessEvent(PEVENT_TRACE pEvent);

void wmain(void)
{
    ULONG status = ERROR_SUCCESS;
    EVENT_TRACE_LOGFILE trace;
    TRACE_LOGFILE_HEADER* pHeader = &trace.LogfileHeader;
    TRACEHANDLE hTrace = 0;
```

```

// Identify the log file from which you want to consume events
// and the callbacks used to process the events and buffers.

ZeroMemory(&trace, sizeof(EVENT_TRACE_LOGFILE));
trace.LogFileName = (LPWSTR) LOGFILE_PATH;
trace.EventCallback = (PEVENT_CALLBACK) (ProcessEvent);

hTrace = OpenTrace(&trace);
if ((TRACEHANDLE)INVALID_HANDLE_VALUE == hTrace)
{
    wprintf(L"OpenTrace failed with %lu\n", GetLastError());
    goto cleanup;
}

g_bUserMode = pHeader->LogFileMode & EVENT_TRACE_PRIVATE_LOGGER_MODE;

if (pHeader->TimerResolution > 0)
{
    g_TimerResolution = pHeader->TimerResolution / 10000;
}

status = ProcessTrace(&hTrace, 1, 0, 0);
if (status != ERROR_SUCCESS && status != ERROR_CANCELLED)
{
    wprintf(L"ProcessTrace failed with %lu\n", status);
    goto cleanup;
}

cleanup:

if ((TRACEHANDLE)INVALID_HANDLE_VALUE != hTrace)
{
    status = CloseTrace(hTrace);
}
}

VOID WINAPI ProcessEvent(PEVENT_TRACE pEvent)
{
    ULONG64 CPUProcessUnits = 0;
    ULONG CPUUnits = 0;
    double CPUTime = 0;

    // Skips the event if it is the event trace header. Log files contain
    // this event
    // but real-time sessions do not. The event contains the same
    // information as
    // the EVENT_TRACE_LOGFILE.LogfileHeader member that you can access when
    // you open
    // the trace.

    if (IsEqualGUID(pEvent->Header.Guid, EventTraceGuid) &&
        pEvent->Header.Class.Type == EVENT_TRACE_TYPE_INFO)

```

```

    ;
    // Skip this event.
}
else
{
    if (IsEqualGUID(CpuUsageEvent, pEvent->Header.Guid))
    {
        // This example assumes that the start and end events are
        paired.

        // If this is the start event type, retrieve the start time
        values from the
        // event; otherwise, retrieve the end time values from the
        event.

        if (pEvent->Header.Class.Type == EVENT_TRACE_TYPE_START)
        {
            // If the session is a private session, use the
ProcessorTime
            // value to calculate the CPU time; otherwise, use the
            // KernelTime and UserTime values.

            if (g_bUserMode) // Private session
            {
                g_StartProcessTime = pEvent->Header.ProcessorTime;
            }
            else // Kernel session
            {
                g_StartKernelTime = pEvent->Header.KernelTime;
                g_StartUserTime = pEvent->Header.UserTime;
            }
        }
        else if (pEvent->Header.Class.Type == EVENT_TRACE_TYPE_END)
        {
            if (g_bUserMode) // Private session
            {
                // Calculate CPU time units used.

                CPUProcessUnits = pEvent->Header.ProcessorTime -
g_StartProcessTime;
                wprintf(L"CPU time units used, %Lu.\n",
CPUProcessUnits);

                // Processor time is in CPU ticks. Convert ticks to
seconds.
                // 1000000000 = nanoseconds in one second.

                CPUTime = (double)(CPUProcessUnits) / 1000000000;
                wprintf(L"Process CPU usage in seconds, %Lf.\n",
CPUTime);
            }
            else // Kernel session
            {
                // Calculate the kernel mode CPU time units used for the
                set of instructions.
            }
        }
    }
}

```

```
        CPUUnits = pEvent->Header.KernelTime -
g_StartKernelTime;
        wprintf(L"CPU time units used (kernel), %d.\n",
CPUUnits);

        // Calculate the kernel mode CPU time in seconds for the
set of instructions.
        // 100 = 100 nanoseconds, 1000000000 = nanoseconds in
one second

        CPUTime = (double)(g_TimerResolution * CPUUnits * 100) /
1000000000;
        wprintf(L"Kernel mode CPU usage in seconds, %Lf.\n",
CPUTime);

        // Calculate user mode CPU time units used for the set
of instructions.

        CPUUnits = pEvent->Header.UserTime - g_StartUserTime;
        wprintf(L"\nCPU time units used (user), %d.\n",
CPUUnits);

        // Calculate the user mode CPU time in seconds for the
set of instructions.
        // 100 = 100 nanoseconds, 1000000000 = nanoseconds in
one second

        CPUTime = (double)(g_TimerResolution * CPUUnits * 100) /
1000000000;
        wprintf(L"User mode CPU usage in seconds, %Lf.\n",
CPUTime);
    }
}
}
}
```

# Event Tracing Reference

Article • 01/07/2021 • 2 minutes to read

You use the following programming elements to write applications that incorporate event tracing:

- [Event Tracing Enumerations](#)
- [Event Tracing Functions](#)
- [Event Tracing Interfaces](#)
- [Event Tracing Structures](#)
- [Event Tracing Constants](#)

For details on samples that use these programming elements, see [Event Tracing Samples](#).

This section also contains information on:

- [Tools](#) that ETW provides
- [MOF class definitions](#) for kernel events
- [MOF class qualifiers](#) used when defining your event classes

## Process ETW traces in .NET code

You can also use the [.NET TraceProcessing API](#) to analyze ETW traces for your applications and other software components. This API is used internally at Microsoft to analyze ETW data produced by the Windows engineering system, and it is also used to power several tables in [Windows Performance Analyzer](#). This API is available as a NuGet package.

For more information, see [this article](#).

# Event Tracing Constants

Article • 01/07/2021 • 2 minutes to read

The following constants are used in event tracing.

| Constant                                   | Description   |
|--|---|
| <a href="#">NT Kernel Logger Constants</a> | Constants defined for system events traced by the NT Kernel Logger event tracing session.   |
| <a href="#">Logging Mode Constants</a>     | Options used by the Log FileMode members of <a href="#">EVENT_TRACE_LOGFILE</a> , <a href="#">EVENT_TRACE_PROPERTIES</a> and <a href="#">TRACE_LOGFILE_HEADER</a> structures. |

# NT Kernel Logger Constants

Article • 08/19/2021 • 2 minutes to read

Use the following constants to identify the NT Kernel Logger session.

| Constant               | Description  |
|------------------------|--|
| SystemTraceControlGuid | The control GUID for the NT Kernel Logger event tracing session. |
| KERNEL_LOGGER_NAME     | The name of the NT Kernel Logger event tracing session.          |

The NT Kernel Logger session is the only session that can accept events from kernel event providers. The NT Kernel Logger session does not accept events from other providers. If you want to capture kernel events and events from other providers, you must use two separate sessions and the consumer would need to merge the events from the log files to provide end-to-end results.

ETW uses the `DEFINE_GUID` macro to define GUIDs. To use `SystemTraceControlGuid` in your code, you must include `#define INITGUID` before including `Evntrace.h`. The compiler will then turn the `DEFINE_GUID` into a constant GUID.

The following values define the possible class GUIDs for kernel events that an NT Kernel Logger session can trace. You can pass the class GUIDs to the [SetTraceCallback](#) function to set up special processing for each event class.

| Class  | GUID  |
|--------|---|
| ALPC   | <pre>DEFINE_GUID ( /* 45d8cccd-539f-4b72-a8b7-5c683142609a */ ALPCGuid,      0x45d8cccd,      0x539f,      0x4b72,      0xa8, 0xb7, 0x5c, 0x68, 0x31, 0x42, 0x60, 0x9a );</pre> |
| DiskIo | <pre>DEFINE_GUID ( /* 3d6fa8d4-fe05-11d0-9dda-00c04fd7ba7c */ DiskIoGuid,     0x3d6fa8d4,     0xfe05,     0x11d0,     0x9d, 0xda, 0x00, 0xc0, 0x4f, 0xd7, 0xba, 0x7c );</pre>   |

| Class                           | GUID  |
|---------------------------------|---|
| HWConfig<br>and<br>SystemConfig | <pre>DEFINE_GUID ( /* 01853a65-418f-4f36-aefc-dc0f1d2fd235 */ EventTraceConfigGuid,     0x01853a65,     0x418f,     0x4f36, 0xae, 0xfc, 0xdc, 0x0f, 0x1d, 0x2f, 0xd2, 0x35 );</pre> |
| FileIo                          | <pre>DEFINE_GUID ( /* 90cbdc39-4a3e-11d1-84f4-0000f80464e3 */ FileIoGuid,     0x90cbdc39,     0x4a3e,     0x11d1,     0x84, 0xf4, 0x00, 0x00, 0xf8, 0x04, 0x64, 0xe3 );</pre>       |
| Image                           | <pre>DEFINE_GUID ( /* 2cb15d1d-5fc1-11d2-abe1-00a0c911f518 */ ImageLoadGuid,     0x2cb15d1d,     0x5fc1,     0x11d2,     0xab, 0xe1, 0x00, 0xa0, 0xc9, 0x11, 0xf5, 0x18 );</pre>    |
| PageFault_V2                    | <pre>DEFINE_GUID ( /* 3d6fa8d3-fe05-11d0-9dda-00c04fd7ba7c */ PageFaultGuid,     0x3d6fa8d3,     0xfe05,     0x11d0,     0x9d, 0xda, 0x00, 0xc0, 0x4f, 0xd7, 0xba, 0x7c );</pre>    |
| PerfInfo                        | <pre>DEFINE_GUID ( /* ce1dbfb4-137e-4da6-87b0-3f59aa102cbc */ PerfInfoGuid,     0xce1dbfb4,     0x137e,     0x4da6,     0x87, 0xb0, 0x3f, 0x59, 0xaa, 0x10, 0x2c, 0xbc );</pre>     |
| Process                         | <pre>DEFINE_GUID ( /* 3d6fa8d0-fe05-11d0-9dda-00c04fd7ba7c */ ProcessGuid,     0x3d6fa8d0,     0xfe05,     0x11d0,     0x9d, 0xda, 0x00, 0xc0, 0x4f, 0xd7, 0xba, 0x7c );</pre>      |

| Class    | GUID  |
|----------|---|
| Registry | <pre>DEFINE_GUID ( /* AE53722E-C863-11d2-8659-00C04FA321A1 */ RegistryGuid,     0xae53722e,     0xc863,     0x11d2,     0x86, 0x59, 0x0, 0xc0, 0x4f, 0xa3, 0x21, 0xa1);</pre> |
| SplitIo  | <pre>DEFINE_GUID ( /* d837ca92-12b9-44a5-ad6a-3a65b3578aa8 */ SplitIoGuid,     0xd837ca92,     0x12b9,     0x44a5,     0xad, 0x6a, 0x3a, 0x65, 0xb3, 0x57, 0x8a, 0xa8);</pre> |
| TcpIp    | <pre>DEFINE_GUID ( /* 9a280ac0-c8e0-11d1-84e2-00c04fb998a2 */ TcpIpGuid,     0x9a280ac0,     0xc8e0,     0x11d1,     0x84, 0xe2, 0x00, 0xc0, 0x4f, 0xb9, 0x98, 0xa2 );</pre>  |
| Thread   | <pre>DEFINE_GUID ( /* 3d6fa8d1-fe05-11d0-9dda-00c04fd7ba7c */ ThreadGuid,     0x3d6fa8d1,     0xfe05,     0x11d0,     0x9d, 0xda, 0x00, 0xc0, 0x4f, 0xd7, 0xba, 0x7c );</pre> |
| UdpIp    | <pre>DEFINE_GUID ( /* bf3a50c5-a9c9-4988-a005-2df0b7c80f80 */ UdpIpGuid,     0xbf3a50c5,     0xa9c9,     0x4988,     0xa0, 0x05, 0x2d, 0xf0, 0xb7, 0xc8, 0x0f, 0x80 );</pre>  |

## Remarks

To use the GUIDs, copy the GUID definitions that you want to use to your source code. You must include #define INITGUID before the definitions you include in your source code, so the compiler will turn the DEFINE\_GUID into a constant GUID. For example,

syntax

```
#define INITGUID

DEFINE_GUID ( /* 3d6fa8d1-fe05-11d0-9dda-00c04fd7ba7c */
    ThreadGuid,
    0x3d6fa8d1,
    0xfe05,
    0x11d0,
    0x9d, 0xda, 0x00, 0xc0, 0x4f, 0xd7, 0xba, 0x7c
);

DEFINE_GUID ( /* 3d6fa8d0-fe05-11d0-9dda-00c04fd7ba7c */
    ProcessGuid,
    0x3d6fa8d0,
    0xfe05,
    0x11d0,
    0x9d, 0xda, 0x00, 0xc0, 0x4f, 0xd7, 0xba, 0x7c
);
```

As an alternative, you can define the constant GUID for the GUID definitions yourself. For example,

syntax

```
static const GUID ThreadGuid =
{ 0x3d6fa8d0, 0xfe05, 0x11d0, { 0x9d, 0xda, 0x00, 0xc0, 0x4f, 0xd7, 0xba,
0x7c } };
```

# Logging Mode Constants

Article • 08/19/2021 • 9 minutes to read

The following constants represent the possible logging modes for an event tracing session.

The constants are used in the `LogFileMode` members of [EVENT\\_TRACE\\_LOGFILE](#), [EVENT\\_TRACE\\_PROPERTIES](#) and [TRACE\\_LOGFILE\\_HEADER](#) structures. These constants are defined in the *Evtrace.h* header file.

| Mode   | Description   |
|--|---|
| <code>EVENT_TRACE_FILE_MODE_NONE</code> (0x00000000)       | Same as <code>EVENT_TRACE_FILE_MODE_SEQUENTIAL</code> with no maximum file size specified.  |
| <code>EVENT_TRACE_FILE_MODE_SEQUENTIAL</code> (0x00000001) | Writes events to a log file sequentially; stops when the file reaches its maximum size. Do not use with <code>EVENT_TRACE_FILE_MODE_CIRCULAR</code> or <code>EVENT_TRACE_FILE_MODE_NEWFILE</code> .   |
| <code>EVENT_TRACE_FILE_MODE_CIRCULAR</code> (0x00000002)   | Writes events to a log file. After the file reaches the maximum size, the oldest events are replaced with incoming events. Note that the contents of the circular log file may appear out of order on multiprocessor computers.<br>Do not use with <code>EVENT_TRACE_FILE_MODE_APPEND</code> , <code>EVENT_TRACE_FILE_MODE_NEWFILE</code> , or <code>EVENT_TRACE_FILE_MODE_SEQUENTIAL</code> .  |
| <code>EVENT_TRACE_FILE_MODE_APPEND</code> (0x00000004)     | Appends events to an existing sequential log file. If the file does not exist, it is created. Use only if you specify <a href="#">system time</a> for the clock resolution, otherwise, <a href="#">ProcessTrace</a> will return events with incorrect time stamps. When using <code>EVENT_TRACE_FILE_MODE_APPEND</code> , the values for <code>BufferSize</code> , <code>NumberOfProcessors</code> , and <code>ClockType</code> must be explicitly provided and must be the same in both the logger and the file being appended.<br>Do not use with <code>EVENT_TRACE_REAL_TIME_MODE</code> , <code>EVENT_TRACE_FILE_MODE_CIRCULAR</code> , <code>EVENT_TRACE_FILE_MODE_NEWFILE</code> , or <code>EVENT_TRACE_PRIVATE_LOGGER_MODE</code> .<br><b>Windows 2000:</b> This value is not supported. |

| Mode  | Description  |
|---|--|
| <code>EVENT_TRACE_FILE_MODE_NEWFILE</code> (0x00000008)     | <p>Automatically switches to a new log file when the file reaches the maximum size. The <code>MaximumFileSize</code> member of <a href="#">EVENT_TRACE_PROPERTIES</a> must be set. The specified file name must be a formatted string (for example, the string contains a %d, such as c:\test%d.etl). Each time a new file is created, a counter is incremented and its value is used, the formatted string is updated, and the resulting string is used as the file name.</p> <p>This option is not allowed for private event tracing sessions and should not be used for NT kernel logger sessions.</p> <p>Do not use with <code>EVENT_TRACE_FILE_MODE_CIRCULAR</code>, <code>EVENT_TRACE_FILE_MODE_APPEND</code> or <code>EVENT_TRACE_FILE_MODE_SEQUENTIAL</code>.</p> <p><b>Windows 2000:</b> This value is not supported.</p> |
| <code>EVENT_TRACE_FILE_MODE_PREALLOCATE</code> (0x00000020) | <p>Reserves <code>EVENT_TRACE_PROPERTIES.MaximumFileSize</code> bytes of disk space for the log file in advance. The file occupies the entire space during logging, for both circular and sequential log files. When you stop the session, the log file is reduced to the size needed. You must set <code>EVENT_TRACE_PROPERTIES.MaximumFileSize</code>. You cannot use the mode for private event tracing sessions.</p> <p><b>Windows 2000:</b> This value is not supported.</p>  |
| <code>EVENT_TRACE_NONSTOPPABLE_MODE</code> (0x00000040)     | <p>The logging session cannot be stopped. This mode is only supported by Autologger. This option is supported on Windows Vista and later.</p> <p>.</p>   |
| <code>EVENT_TRACE_SECURE_MODE</code> (0X00000080)           | <p>Restricts who can log events to the session to those with <a href="#">TRACELOG_LOG_EVENT</a> permission. This option is supported on Windows Vista and later.</p>   |

| Mode   | Description   |
|--|---|
| EVENT_TRACE_REAL_TIME_MODE (0x00000100)      | <p>Delivers the events to consumers in real-time. Events are delivered when the buffers are flushed, not at the time the provider writes the event. You should not enable real-time mode if there are no consumers to consume the events because calls to log events will eventually fail when the buffers become full. Prior to Windows Vista, if the events were not being consumed, the events were discarded. Do not specify more than one real-time consumer in one process on Windows XP or Windows Server 2003. Instead, have one thread consume events and distribute the events to others.</p> <p><b>Prior to Windows Vista:</b> You should not use real-time mode because the supported event rate is much lower than reading from the log file (events may be dropped). Also, the event order is not guaranteed on computers with multiple processors. The real-time mode is more suitable for low-traffic, notification type events.</p> <p>You can combine this mode with other log file modes; however, do not use this mode with EVENT_TRACE_PRIVATE_LOGGER_MODE. Note that if you combine this mode with other log file modes, buffers will be flushed once every second, resulting in partially filled buffers being written to your log file. For example if you use 64k buffers and your logging rate is 1 event every second, the service will write 64k/second to your log file.</p> |
| EVENT_TRACE_DELAY_OPEN_FILE_MODE(0x00000200) | <p>This mode is used to delay opening the log file until an event occurs.</p> <p><b>Note:</b><br/>On Windows Vista or later, this mode is not applicable should not be used.</p>  |

| Mode  | Description   |
|---|---|
| <b>EVENT_TRACE_BUFFERING_MODE</b> (0x00000400)      | <p>This mode writes events to a circular memory buffer. Events written beyond the total size of the buffer evict the oldest events still remaining in the buffer. The size of this memory buffer is the product of <b>MinimumBuffers</b> and <b>BufferSize</b> (see <a href="#">EVENT_TRACE_PROPERTIES</a>). As a consequence of this formula, any buffer that uses <b>EVENT_TRACE_BUFFERING_MODE</b> will ignore the <b>MaximumBuffers</b> value.</p> <p>Events are not written to a log file or delivered in real-time, and ETW does not flush the buffers. To get a snapshot of the buffer, call the <a href="#">FlushTrace</a> function.</p> <p>This mode is particularly useful for debugging device drivers in conjunction with the ability to view the contents of in-memory buffers with the <a href="#">WMITrace</a> kernel debugger extension.</p> <p>Do not use with <b>EVENT_TRACE_FILE_MODE_SEQUENTIAL</b>, <b>EVENT_TRACE_FILE_MODE_CIRCULAR</b>, <b>EVENT_TRACE_FILE_MODE_APPEND</b>, <b>EVENT_TRACE_FILE_MODE_NEWFILE</b>, or <b>EVENT_TRACE_REAL_TIME_MODE</b>.</p>  |
| <b>EVENT_TRACE_PRIVATE_LOGGER_MODE</b> (0x00000800) | <p>Creates a user-mode event tracing session that runs in the same process as its event trace provider. The memory for buffers comes from the process's memory. Processes that do not require data from the kernel can eliminate the overhead associated with kernel-mode transitions by using a private event tracing session.</p> <p>If the provider is registered by multiple processes, ETW appends the process identifier to the log file name to create a unique log file name. For example, if the controller specifies the log file names as c:\mylogs\myprivatelog.etl, ETW creates the log file as c:\mylogs\myprivatelog.etl_nnnn, where nnnn is the process identifier. The process identifier is not appended to the first process that registers the provider, it is appended to only the subsequent processes that register the provider.</p> <p>Private event tracing sessions have the following limitations:</p> <ul style="list-style-type: none"> <li>• A private session can record events only for the threads of the process in which it is executing.</li> <li>• There can be up to eight private session per process.</li> <li>• Private sessions cannot be used with real-time delivery.</li> <li>• Events that are generated by a private session do not include execution time for</li> </ul> |

| Mode  | Description  |
|---|--|
|   | kernel-mode versus user-mode instructions, or thread-level detail of the CPU time used.  |
|   | <p>Process ID filters and executable name filters can now be passed in to session control APIs when system wide private loggers are started. For the best results in cross process scenarios, the same filters should be passed to every control operation during the session, including provider enable/diable calls. Note that the filters have the same format as those consumed by <a href="#">EnableTraceEx2</a>.</p> <p>You can use this mode in conjunction with the <code>EVENT_TRACE_PRIVATE_IN_PROC</code> mode.</p> <p><b>Prior to Windows 10, version 1703:</b> Only LocalSystem, the administrator, and users in the administrator group that run in an elevated process can create a private session. If you include the <code>EVENT_TRACE_PRIVATE_IN_PROC</code> flag, any user can create an in-process private session. Also, in prior versions of Windows, there can only be one private session per process (unless the <code>EVENT_TRACE_PRIVATE_IN_PROC</code> mode is also specified, in which case you can create up to three in-process private sessions).</p> <p><b>Prior to Windows Vista:</b> Users in the Performance Log Users group could also create a private session.</p> |
|   | <p>Do not use with <code>EVENT_TRACE_REAL_TIME_MODE</code>.</p> <p><b>Prior to Windows 7 and Windows Server 2008 R2:</b></p> <p>Do not use with <code>EVENT_TRACE_FILE_MODE_NEWFILE</code>.</p>  |
| <code>EVENT_TRACE_ADD_HEADER_MODE</code> (0x00001000)     | <p>This option adds a header to the log file.</p> <p><b>Note:</b><br/>On Windows Vista or later, this mode is not applicable should not be used.</p>   |
| <code>EVENT_TRACE_USE_KBYTES_FOR_SIZE</code> (0x00002000) | <p>Use kilobytes as the unit of measure for specifying the size of a file. The default unit of measure is megabytes. This mode applies to the <code>MaxFileSize</code> registry value for an <a href="#">AutoLogger</a> session and the <code>MaximumFileSize</code> member of <code>EVENT_TRACE_PROPERTIES</code>. This option is supported on Windows Vista and later.</p>   |

| Mode  | Description  |
|---|--|
| <code>EVENT_TRACE_USE_GLOBAL_SEQUENCE</code> (0x00004000)     | <p>Uses sequence numbers that are unique across event tracing sessions. This mode only applies to events logged using the <a href="#">TraceMessage</a> function. For more information, see <a href="#">TraceMessage</a> for usage details.</p> <p><code>EVENT_TRACE_USE_GLOBAL_SEQUENCE</code> and <code>EVENT_TRACE_USE_LOCAL_SEQUENCE</code> are mutually exclusive.</p> <p><b>Windows 2000:</b> This value is not supported.</p>                |
| <code>EVENT_TRACE_USE_LOCAL_SEQUENCE</code> (0x00008000)      | <p>Uses sequence numbers that are unique only for an individual event tracing session. This mode only applies to events logged using the <a href="#">TraceMessage</a> function. For more information, see <a href="#">TraceMessage</a> for usage details.</p> <p><code>EVENT_TRACE_USE_GLOBAL_SEQUENCE</code> and <code>EVENT_TRACE_USE_LOCAL_SEQUENCE</code> are mutually exclusive.</p> <p><b>Windows 2000:</b> This value is not supported.</p> |
| <code>EVENT_TRACE_RELOG_MODE</code> (0x00010000)              | <p>Logs the event without including <a href="#">EVENT_TRACE_HEADER</a>.</p> <p><b>Note:</b><br/>This mode should not be used. It is reserved for internal use.</p> <p><b>Windows 2000:</b> This value is not supported.</p>  |
| <code>EVENT_TRACE_PRIVATE_IN_PROC</code> (0x00020000)         | <p>Use in conjunction with the <code>EVENT_TRACE_PRIVATE_LOGGER_MODE</code> mode to start a private session. This mode enforces that only the process that registered the provider GUID can start the logger session with that GUID. You can create up to three in-process private sessions per process.</p> <p>This option is supported on Windows Vista and later.</p>   |
| <code>EVENT_TRACE_MODE_RESERVED</code> (0x00100000)           | <p>This option is used to signal heap and critical section tracing. This option is supported on Windows Vista and later.</p>   |
| <code>EVENT_TRACE_STOP_ON_HYBRID_SHUTDOWN</code> (0x00400000) | <p>This option stops logging on hybrid shutdown. If neither <code>EVENT_TRACE_STOP_ON_HYBRID_SHUTDOWN</code> or <code>EVENT_TRACE_PERSIST_ON_HYBRID_SHUTDOWN</code> is specified, ETW will chose a default based on whether the caller is coming from Session 0 or not. This option is supported on Windows 8 and Windows Server 2012.</p>   |

| Mode   | Description  |
|--|--|
| <code>EVENT_TRACE_PERSIST_ON_HYBRID_SHUTDOWN</code> (0x00800000) | <p>This option continues logging on hybrid shutdown. If neither <code>EVENT_TRACE_STOP_ON_HYBRID_SHUTDOWN</code> or <code>EVENT_TRACE_PERSIST_ON_HYBRID_SHUTDOWN</code> is specified, ETW will chose a default based on whether the caller is coming from Session 0 or not. This option is supported on Windows 8 and Windows Server 2012.</p>   |
| <code>EVENT_TRACE_USE_PAGED_MEMORY</code> (0x01000000)           | <p>Uses paged memory. This setting is recommended so that events do not use up the nonpaged memory. Nonpaged buffers use nonpaged memory for buffer space. Because nonpaged buffers are never paged out, a logging session performs well. Using pageable buffers is less resource-intensive. Kernel-mode providers and system loggers cannot log events to sessions that specify this logging mode.</p> <p>This mode is ignored if <code>EVENT_TRACE_PRIVATE_LOGGER_MODE</code> is set. You cannot use this mode with the NT Kernel Logger.</p> <p><b>Windows 2000:</b> This value is not supported.</p> |
| <code>EVENT_TRACE_SYSTEM_LOGGER_MODE</code> (0x02000000)         | <p>This option will receive events from SystemTraceProvider. If the <code>StartTraceProperties</code> parameter <code>LogFileMode</code> includes this flag, the logger will be a system logger. This option is supported on Windows 8 and Windows Server 2012.</p>  |
| <code>EVENT_TRACE_INDEPENDENT_SESSION_MODE</code> (0x08000000)   | <p>Indicates that a logging session should not be affected by <code>EventWrite</code> failures in other sessions. Without this flag, if an event cannot be published to one of the sessions that a provider is enabled to, the event will not get published to any of the sessions. When this flag is set, a failure to write an event to one session will not cause the <code>EventWrite</code> function to return an error code in other sessions. Do not use with <code>EVENT_TRACE_PRIVATE_LOGGER_MODE</code>. This option is supported on Windows 8.1, Windows Server 2012 R2, and later.</p>       |

| Mode   | Description  |
|--|--|
| <b>EVENT_TRACE_NO_PER_PROCESSOR_BUFFERING</b> (0x10000000) | <p>Writes events that were logged on different processors to a common buffer. Using this mode can eliminate the issue of events appearing out of order when events are being published on different processors using system time. This mode can also eliminate the issue with circular logs appearing to drop events on multiple processor computers.</p> <p>If you do not use this mode and you use system time, the events may appear out of order on multiple processor computers. This is because ETW buffers are associated with a processor instead of a thread. As a result, if a thread is switched from one CPU to another, the buffer associated with the latter CPU can be flushed to disk before the one associated with the former CPU.</p> <p>If you expect a high volume of events (for example, more than 1,000 events per second), you should not use this mode.</p> <p>Note that the processor number is not included with the event.</p> <p>This option is supported on Windows 7, Windows Server 2008 R2, and later.</p> |
| <b>EVENT_TRACE_ADDTO_TRIAGE_DUMP</b> (0x80000000)          | <p>This option adds ETW buffers to triage dumps. This option is supported on Windows 8 and Windows Server 2012.</p>  |

# Event Tracing Macros

Article • 01/07/2021 • 2 minutes to read

ETW defines the following convenience macros that you can use to access members of the EVENT\_DESCRIPTOR structure.

[EventDataDescCreate](#)  
[EventDescCreate](#)  
[EventDescGetChannel](#)  
[EventDescGetId](#)  
[EventDescGetKeyword](#)  
[EventDescGetLevel](#)  
[EventDescGetOpcode](#)  
[EventDescGetTask](#)  
[EventDescGetVersion](#)  
[EventDescOrKeyword](#)  
[EventDescSetChannel](#)  
[EventDescSetId](#)  
[EventDescSetKeyword](#)  
[EventDescSetLevel](#)  
[EventDescSetOpcode](#)  
[EventDescSetTask](#)  
[EventDescSetVersion](#)  
[EventDescZero](#)

# EventDataDescCreate function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Sets the values of an [EVENT\\_DATA\\_DESCRIPTOR](#).

## Syntax

C++

```
EVNTPROV_PFORCEINLINE VOID EventDataDescCreate(
    [out] PEVENT_DATA_DESCRIPTOR EventDescriptor,
    [in]  const VOID             *DataPtr,
    [in]  ULONG                 DataSize
);
```

## Parameters

[out] EventDescriptor

The data descriptor whose member values are set to those of the remaining parameters. For details, see [EVENT\\_DATA\\_DESCRIPTOR](#).

[in] DataPtr

A pointer to the event data. This value is used to set the *Ptr* member of the descriptor.

*DataPtr* parameter may be **NULL** if and only if *DataSize* is 0.

[in] DataSize

The size (in bytes) of the event data. The value is used to set the *Size* member of the descriptor.

## Return value

This function does not return a value.

## Remarks

This is a convenience macro for setting the members of the [EVENT\\_DATA\\_DESCRIPTOR](#) structure. Note that if you initialize the members yourself without calling [EventDataDescCreate](#), you should set `Ptr = (UINT_PTR)DataPtr`, and you **must** initialize the *Reserved* field (e.g. set it to 0),

## Requirements

|                          |   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps   UWP apps]       |
| Minimum supported server | Windows Server 2008 [desktop apps   UWP apps] |
| Target Platform          | Windows                                       |
| Header                   | evntprov.h                                    |

# EventDescCreate function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Sets the values of an event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE VOID EventDescCreate(
    [out] PEVENT_DESCRIPTOR EventDescriptor,
    [in]  USHORT           Id,
    [in]  UCHAR            Version,
    [in]  UCHAR            Channel,
    [in]  UCHAR            Level,
    [in]  USHORT           Task,
    [in]  UCHAR            Opcode,
    [in]  UONGLONG         Keyword
);
```

## Parameters

[out] EventDescriptor

Event descriptor whose member values are set to those of the remaining parameters.  
For details, see [EVENT\\_DESCRIPTOR](#).

[in] Id

Event identifier. The value is used to set the **Id** member of [EVENT\\_DESCRIPTOR](#).

[in] Version

Version of the event. The value is used to set the **Version** member of [EVENT\\_DESCRIPTOR](#).

[in] Channel

The category of events to which this event belongs. The value is used to set the **Channel** member of [EVENT\\_DESCRIPTOR](#).

[in] Level

Specifies the severity of the event. The value is used to set the **Level** member of [EVENT\\_DESCRIPTOR](#).

[in] Task

Identifies a logical component of the application whose events you want to enable. The value is used to set the **Task** member of [EVENT\\_DESCRIPTOR](#).

[in] Opcode

Operation being performed at the time the event was written. The value is used to set the **Opcode** member of [EVENT\\_DESCRIPTOR](#).

[in] Keyword

Bitmask that further defines the category of events to which the event belongs. The value is used to set the **Keyword** member of [EVENT\\_DESCRIPTOR](#).

## Return value

This function does not return a value.

## Remarks

This is a convenience macro for setting the members of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

|                          |   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |
| Header                   | evntprov.h                              |

## See also

[EventDescZero](#)

# EventDescGetChannel function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Retrieves the channel from the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE UCHAR EventDescGetChannel(
    [in] PCEVENT_DESCRIPTOR EventDescriptor
);
```

## Parameters

[in] EventDescriptor

Event descriptor from which to retrieve the channel. See [EVENT\\_DESCRIPTOR](#).

## Return value

Channel that defines the category of events to which this event belongs.

## Remarks

This is a convenience macro for retrieving the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

|                          |   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |

Header

evntprov.h

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescGetId function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Retrieves the event identifier from the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE USHORT EventDescGetId(  
    [in] PCEVENT_DESCRIPTOR EventDescriptor  
) ;
```

## Parameters

[in] `EventDescriptor`

Event descriptor from which to retrieve the event identifier. See [EVENT\\_DESCRIPTOR](#).

## Return value

The event identifier.

## Remarks

This is a convenience macro for retrieving the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

| Minimum supported client | Windows Vista [desktop apps only]       |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |
| Header                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescGetKeyword function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Retrieves the keyword from the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE ULONGLONG EventDescGetKeyword(
    [in] PCEVENT_DESCRIPTOR EventDescriptor
);
```

## Parameters

[in] EventDescriptor

Event descriptor from which to retrieve the keyword. See [EVENT\\_DESCRIPTOR](#).

## Return value

Keyword that is a bitmask that further defines the category of events to which the event belongs.

## Remarks

This is a convenience macro for retrieving the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

| Minimum supported client | Windows Vista [desktop apps only]       |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |

Header

evntprov.h

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescGetLevel function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Retrieves the severity level from the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE UCHAR EventDescGetLevel(
    [in] PCEVENT_DESCRIPTOR EventDescriptor
);
```

## Parameters

[in] `EventDescriptor`

Event descriptor from which to retrieve the severity level. See [EVENT\\_DESCRIPTOR](#).

## Return value

Severity level that indicates the verboseness with which to log the event.

## Remarks

This is a convenience macro for retrieving the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

| Minimum supported client | Windows Vista [desktop apps only]       |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |
| Header                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescGetOpcode function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Retrieves the operation code from the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE UCHAR EventDescGetOpcode(
    [in] PCEVENT_DESCRIPTOR EventDescriptor
);
```

## Parameters

[in] EventDescriptor

Event descriptor from which to retrieve the operation code. See [EVENT\\_DESCRIPTOR](#).

## Return value

Operation code that identifies the operation being performed at the time the event was written.

## Remarks

This is a convenience macro for retrieving the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

| Minimum supported client | Windows Vista [desktop apps only]       |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |

Header

evntprov.h

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescGetTask function (evntprov.h)

Article • 10/13/2012 minutes to read

Retrieves the task from the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE USHORT EventDescGetTask(
    [in] PCEVENT_DESCRIPTOR EventDescriptor
);
```

## Parameters

[in] EventDescriptor

Event descriptor from which to retrieve the task. See [EVENT\\_DESCRIPTOR](#).

## Return value

Task that identifies the logical component of the application whose events you want to enable.

## Remarks

This is a convenience macro for retrieving the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

| Minimum supported client | Windows Vista [desktop apps only]       |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |
| Header                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescGetVersion function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Retrieves the version from the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE UCHAR EventDescGetVersion(
    [in] PCEVENT_DESCRIPTOR EventDescriptor
);
```

## Parameters

[in] EventDescriptor

Event descriptor from which to retrieve the version. See [EVENT\\_DESCRIPTOR](#).

## Return value

Version that identifies the revision level of the event definition.

## Remarks

This is a convenience macro for retrieving the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

|                          |   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |

Header

evntprov.h

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescOrKeyword function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Adds another keyword to the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE PEVENT_DESCRIPTOR EventDescOrKeyword(
    [in] PEVENT_DESCRIPTOR EventDescriptor,
    [in] ULONGLONG          Keyword
);
```

## Parameters

[in] EventDescriptor

Event descriptor to modify. See [EVENT\\_DESCRIPTOR](#).

[in] Keyword

New keyword to add to the current keyword bitmask.

## Return value

The modified event descriptor.

## Remarks

This is a convenience macro for setting the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

Minimum supported client

Windows Vista [desktop apps only]

|                          |   |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |
| Header                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescSetChannel function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Sets the **Channel** member of the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE PEVENT_DESCRIPTOR EventDescSetChannel(
    [in] PEVENT_DESCRIPTOR EventDescriptor,
    [in] UCHAR             Channel
);
```

## Parameters

[in] **EventDescriptor**

Event descriptor to modify. See [EVENT\\_DESCRIPTOR](#).

[in] **Channel**

Channel that defines the category of events to which this event belongs.

## Return value

The modified event descriptor.

## Remarks

This is a convenience macro for setting the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

Minimum supported client

Windows Vista [desktop apps only]

|                          |   |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |
| Header                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescSetId function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Sets the **Id** member of the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE PEVENT_DESCRIPTOR EventDescSetId(
    [in] PEVENT_DESCRIPTOR EventDescriptor,
    [in] USHORT           Id
);
```

## Parameters

[in] **EventDescriptor**

Event descriptor to modify. See [EVENT\\_DESCRIPTOR](#).

[in] **Id**

The event identifier.

## Return value

The modified event descriptor.

## Remarks

This is a convenience macro for setting the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

|                          |   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

|                 |            |
|-----------------|------------|
|                 |            |
| Target Platform | Windows    |
| Header          | evntprov.h |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescSetKeyword function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Sets the **Keyword** member of the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE PEVENT_DESCRIPTOR EventDescSetKeyword(
    [in] PEVENT_DESCRIPTOR EventDescriptor,
    [in] ULONGLONG          Keyword
);
```

## Parameters

[in] `EventDescriptor`

Event descriptor to modify. See [EVENT\\_DESCRIPTOR](#).

[in] `Keyword`

Keyword that is a bitmask that further defines the category of events to which the event belongs.

## Return value

The modified event descriptor.

## Remarks

This is a convenience macro for setting the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

|                                 |   |
|---------------------------------|---|
|                                 |   |
| <b>Minimum supported client</b> | Windows Vista [desktop apps only]       |
| <b>Minimum supported server</b> | Windows Server 2008 [desktop apps only] |
| <b>Target Platform</b>          | Windows                                 |
| <b>Header</b>                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescSetLevel function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Sets the **Level** member of the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE PEVENT_DESCRIPTOR EventDescSetLevel(
    [in] PEVENT_DESCRIPTOR EventDescriptor,
    [in] UCHAR             Level
);
```

## Parameters

[in] **EventDescriptor**

Event descriptor to modify. See [EVENT\\_DESCRIPTOR](#).

[in] **Level**

Severity level that indicates the verboseness with which to log the event. For details, see the **Level** parameter of [EnableTraceEx](#).

## Return value

The modified event descriptor.

## Remarks

This is a convenience macro for setting the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

Minimum supported client

Windows Vista [desktop apps only]

|                                 |   |
|---------------------------------|---|
|                                 |   |
| <b>Minimum supported server</b> | Windows Server 2008 [desktop apps only] |
| <b>Target Platform</b>          | Windows                                 |
| <b>Header</b>                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescSetOpcode function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Sets the **Opcode** member of the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE PEVENT_DESCRIPTOR EventDescSetOpcode(  
    [in] PEVENT_DESCRIPTOR EventDescriptor,  
    [in] UCHAR             Opcode  
)
```

## Parameters

[in] **EventDescriptor**

Event descriptor to modify. See [EVENT\\_DESCRIPTOR](#).

[in] **Opcode**

Operation code that identifies the operation being performed at the time the event was written.

## Return value

The modified event descriptor.

## Remarks

This is a convenience macro for setting the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

|                                 |   |
|---------------------------------|---|
|                                 |   |
| <b>Minimum supported client</b> | Windows Vista [desktop apps only]       |
| <b>Minimum supported server</b> | Windows Server 2008 [desktop apps only] |
| <b>Target Platform</b>          | Windows                                 |
| <b>Header</b>                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescSetTask function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Sets the **Task** member of the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE PEVENT_DESCRIPTOR EventDescSetTask(
    [in] PEVENT_DESCRIPTOR EventDescriptor,
    [in] USHORT             Task
);
```

## Parameters

[in] **EventDescriptor**

Event descriptor to modify. See [EVENT\\_DESCRIPTOR](#).

[in] **Task**

Task that identifies the logical component of the application whose events you want to enable.

## Return value

The modified event descriptor.

## Remarks

This is a convenience macro for setting the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

Minimum supported client

Windows Vista [desktop apps only]

|                                 |   |
|---------------------------------|---|
|                                 |   |
| <b>Minimum supported server</b> | Windows Server 2008 [desktop apps only] |
| <b>Target Platform</b>          | Windows                                 |
| <b>Header</b>                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescSetVersion function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Sets the **Version** member of the event descriptor.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE PEVENT_DESCRIPTOR EventDescSetVersion(
    [in] PEVENT_DESCRIPTOR EventDescriptor,
    [in] UCHAR             Version
);
```

## Parameters

[in] `EventDescriptor`

Event descriptor to modify. See [EVENT\\_DESCRIPTOR](#).

[in] `Version`

Version that identifies the revision level of the event definition. The first version of an event is zero.

## Return value

The modified event descriptor.

## Remarks

This is a convenience macro for setting the member of the [EVENT\\_DESCRIPTOR](#) structure.

## Requirements

|                                 |   |
|---------------------------------|---|
|                                 |   |
| <b>Minimum supported client</b> | Windows Vista [desktop apps only]       |
| <b>Minimum supported server</b> | Windows Server 2008 [desktop apps only] |
| <b>Target Platform</b>          | Windows                                 |
| <b>Header</b>                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# EventDescZero function (evntprov.h)

Article • 10/13/2021 2 minutes to read

Initializes an event descriptor to zero.

## Syntax

C++

```
EVNTPROV_PFORCEINLINE VOID EventDescZero(
    [out] PEVENT_DESCRIPTOR EventDescriptor
);
```

## Parameters

[out] EventDescriptor

The event descriptor. See [EVENT\\_DESCRIPTOR](#).

## Return value

This function does not return a value.

## Remarks

This is a convenience macro for initializing the memory of the [EVENT\\_DESCRIPTOR](#) structure to zero.

## Requirements

| Minimum supported client | Windows Vista [desktop apps only]       |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |
| Target Platform          | Windows                                 |
| Header                   | evntprov.h                              |

## See also

[EVENT\\_DESCRIPTOR](#)

# Event Tracing

Article • 01/24/2023 16 minutes to read

Overview of the Event Tracing technology.

To develop Event Tracing, you need these headers:

- [evntcons.h](#)
- [evnprov.h](#)
- [evntrace.h](#)
- [relogger.h](#)
- [securitybaseapi.h](#)
- [tdh.h](#)

For programming guidance for this technology, see:

- [Event Tracing](#)

## Enumerations

|  |
|--|
|  |
| <a href="#">_TDH_IN_TYPE</a>   |
| Defines the supported [in] types for a trace data helper (TDH).                      |
| <a href="#">_TDH_OUT_TYPE</a>  |
| Defines the supported [out] types for a trace data helper (TDH).                     |
| <a href="#">DECODING_SOURCE</a>  |
| Defines the source of the event data.  |
| <a href="#">ETW_PROCESS_HANDLE_INFO_TYPE</a>   |
| Specifies the operation that will be performed on a trace processing session.        |
| <a href="#">ETW_PROCESS_TRACE_MODES</a>  |
| Specifies the supported process trace modes.   |
| <a href="#">ETW_PROVIDER_TRAIT_TYPE</a>  |
| Specifies the types of Provider Traits supported by Event Tracing for Windows (ETW). |

## [EVENT\\_FIELD\\_TYPE](#)

Defines the provider information to retrieve.

## [EVENT\\_INFO\\_CLASS](#)

The EVENT\_INFO\_CLASS enumeration type is used with the EventSetInformation function to specify the configuration operation to be performed on an ETW event provider registration.

## [EVENTSECURITYOPERATION](#)

Defines what component of the security descriptor that the EventAccessControl function modifies.

## [MAP\\_FLAGS](#)

Defines constant values that indicate if the map is a value map, bitmap, or pattern map.

## [MAP\\_VALUETYPE](#)

Defines if the value map value is in a ULONG data type or a string.

## [PAYLOAD\\_OPERATOR](#)

Defines the supported payload operators for a trace data helper (TDH).

## [PROPERTY\\_FLAGS](#)

Defines if the property is contained in a structure or array.

## [TDH\\_CONTEXT\\_TYPE](#)

Defines the context type.

## [TEMPLATE\\_FLAGS](#)

Defines constant values that indicates the layout of the event data.

## [TRACE\\_QUERY\\_INFO\\_CLASS](#)

Used with EnumerateTraceGuidsEx and TraceSetInformation to specify a type of trace information.

# Functions

|   |
|---|
|   |
| <p><a href="#">AddLogFileTraceStream</a></p> <p>Adds a new logfile-based ETW trace stream to the relogger.</p>  |
| <p><a href="#">AddRealtimeTraceStream</a></p> <p>Adds a new real-time ETW trace stream to the relogger.</p>   |
| <p><a href="#">Cancel</a></p> <p>Terminates the relogging process.</p>  |
| <p><a href="#">Clone</a></p> <p>Creates a duplicate copy of an event.</p>   |
| <p><a href="#">CloseTrace</a></p> <p>The CloseTrace function closes a trace processing session that was created with OpenTrace.</p>   |
| <p><a href="#">ControlTraceA</a></p> <p>The ControlTraceA (ANSI) function (evntrace.h) flushes, queries, updates, or stops the specified event tracing session.</p>   |
| <p><a href="#">ControlTraceW</a></p> <p>The ControlTraceW (Unicode) function (evntrace.h) flushes, queries, updates, or stops the specified event tracing session.</p>  |
| <p><a href="#">CreateEventInstance</a></p> <p>Generates a new event.</p>  |
| <p><a href="#">CreateTraceInstanceld</a></p> <p>A RegisterTraceGuids-based ("Classic") event provider uses the CreateTraceInstanceld function to create a unique transaction identifier and map it to a registration handle. The provider can then use the transaction identifier when calling the TraceEventInstance function.</p> |
| <p><a href="#">CveEventWrite</a></p> <p>A tracing function for publishing events when an attempted security vulnerability exploit is detected in your user-mode application.</p>  |
| <p><a href="#">DECLSPEC_XFGVIRT</a></p>   |

|   |
|---|
|   |
| <p><a href="#">EMI_MAP_FORMAT</a></p> <p>Macro that retrieves the event map format.</p>   |
| <p><a href="#">EMI_MAP_INPUT</a></p> <p>Macro that retrieves the event map input.</p>   |
| <p><a href="#">EMI_MAP_NAME</a></p> <p>Macro that retrieves the event map name.</p>   |
| <p><a href="#">EMI_MAP_OUTPUT</a></p> <p>Macro that retrieves the event map output.</p>   |
| <p><a href="#">EnableTrace</a></p> <p>A trace session controller calls EnableTrace to configure how an ETW event provider logs events to a trace session. The EnableTraceEx2 function supersedes this function.</p>     |
| <p><a href="#">EnableTraceEx</a></p> <p>A trace session controller calls EnableTraceEx to configure how an ETW event provider logs events to a trace session. The EnableTraceEx2 function supersedes this function.</p> |
| <p><a href="#">EnableTraceEx2</a></p> <p>A trace session controller calls EnableTraceEx2 to configure how an ETW event provider logs events to a trace session.</p>   |
| <p><a href="#">EnumerateTraceGuids</a></p> <p>Retrieves information about event trace providers that are currently running on the computer. The EnumerateTraceGuidsEx function supersedes this function.</p>            |
| <p><a href="#">EnumerateTraceGuidsEx</a></p> <p>Retrieves information about event trace providers that are currently running on the computer.</p>   |
| <p><a href="#">EtwGetTraitFromProviderTraits</a></p>  |
| <p><a href="#">EventAccessControl</a></p> <p>Adds or modifies the permissions of the specified provider or session.</p>   |

## [EventAccessQuery](#)

Retrieves the permissions for the specified controller or provider.

## [EventAccessRemove](#)

Removes the permissions defined in the registry for the specified provider or session.

## [EventActivityIdControl](#)

Creates, queries, and sets activity identifiers for use in ETW events.

## [EventDataDescCreate](#)

Sets the values of an EVENT\_DATA\_DESCRIPTOR.

## [EventDescCreate](#)

Sets the values of an event descriptor.

## [EventDescGetChannel](#)

Retrieves the channel from the event descriptor.

## [EventDescGetId](#)

Retrieves the event identifier from the event descriptor.

## [EventDescGetKeyword](#)

Retrieves the keyword from the event descriptor.

## [EventDescGetLevel](#)

Retrieves the severity level from the event descriptor.

## [EventDescGetOpcode](#)

Retrieves the operation code from the event descriptor.

## [EventDescGetTask](#)

Retrieves the task from the event descriptor.

## [EventDescGetVersion](#)

Retrieves the version from the event descriptor.

|   |
|---|
|   |
| <a href="#">EventDescOrKeyword</a>  |
| Adds another keyword to the event descriptor.   |
| <a href="#">EventDescSetChannel</a>   |
| Sets the Channel member of the event descriptor.  |
| <a href="#">EventDescSetId</a>  |
| Sets the Id member of the event descriptor.   |
| <a href="#">EventDescSetKeyword</a>   |
| Sets the Keyword member of the event descriptor.  |
| <a href="#">EventDescSetLevel</a>   |
| Sets the Level member of the event descriptor.  |
| <a href="#">EventDescSetOpcode</a>  |
| Sets the Opcode member of the event descriptor.   |
| <a href="#">EventDescSetTask</a>  |
| Sets the Task member of the event descriptor.   |
| <a href="#">EventDescSetVersion</a>   |
| Sets the Version member of the event descriptor.  |
| <a href="#">EventDescZero</a>   |
| Initializes an event descriptor to zero.  |
| <a href="#">EventEnabled</a>  |
| Determines whether an event provider should generate a particular event based on the event's EVENT_DESCRIPTOR.  |
| <a href="#">EventProviderEnabled</a>  |
| Determines whether an event provider should generate a particular event based on the event's Level and Keyword. |
| <a href="#">EventRegister</a>   |
| Registers an ETW event provider, creating a handle that can be used to write ETW events.                        |

|   |
|---|
|   |
| <p><a href="#">EventSetInformation</a></p> <p>Configures an ETW event provider.</p>   |
| <p><a href="#">EventUnregister</a></p> <p>Unregisters an ETW event provider.</p>  |
| <p><a href="#">EventWrite</a></p> <p>Writes an ETW event that uses the current thread's activity ID.</p>  |
| <p><a href="#">EventWriteEx</a></p> <p>Writes an ETW event with an activity ID, an optional related activity ID, session filters, and special options.</p>  |
| <p><a href="#">EventWriteString</a></p> <p>Writes an ETW event that contains a string as its data. This function should not be used.</p>  |
| <p><a href="#">EventWriteTransfer</a></p> <p>Writes an ETW event with an activity ID and an optional related activity ID.</p>   |
| <p><a href="#">FlushTraceA</a></p> <p>The FlushTraceA (ANSI) function (evntrace.h) causes an event tracing session to immediately deliver buffered events for the specified session.</p>  |
| <p><a href="#">FlushTraceW</a></p> <p>The FlushTraceW (Unicode) function (evntrace.h) causes an event tracing session to immediately deliver buffered events for the specified session.</p>   |
| <p><a href="#">GetEventProcessorIndex</a></p>   |
| <p><a href="#">GetEventRecord</a></p> <p>Retrieves the event record that describes an event.</p>  |
| <p><a href="#">GetTraceEnableFlags</a></p> <p>A RegisterTraceGuids-based ("Classic") event provider uses the GetTraceEnableFlags function to retrieve the enable flags specified by the trace controller to indicate which category of events to trace. Providers call this function from their ControlCallback function.</p> |

## [GetTraceEnableLevel](#)

A RegisterTraceGuids-based ("Classic") event provider uses the GetTraceEnableLevel function to retrieve the enable level specified by the trace controller to indicate which level of events to trace. Providers call this function from their ControlCallback function.

## [GetTraceLoggerHandle](#)

A RegisterTraceGuids-based ("Classic") event provider uses the GetTraceLoggerHandle function to retrieve the handle of the event tracing session to which it should write events. Providers call this function from their ControlCallback function.

## [GetUserContext](#)

Retrieves the user context associated with the stream to which the event belongs.

## [Inject](#)

Injects a non-system-generated event into the event stream being written to the output trace logfile.

## [OnBeginProcessTrace](#)

Indicates that a trace is about to begin so that relogging can be started.

## [OnEvent](#)

Indicates that an event has been received on the trace streams associated with a relogger.

## [OnFinalizeProcessTrace](#)

Indicates that a trace is about to end so that relogging can be finalized.

## [OpenTraceA](#)

The OpenTraceA (ANSI) function (evntrace.h) opens an ETW trace processing handle for consuming events from an ETW real-time trace session or an ETW log file.

## [OpenTraceFromBufferStream](#)

Creates a trace processing session that is not directly attached to any file or active session.

## [OpenTraceFromFile](#)

Creates a trace processing session to process a Tracelog .etl file.

## [OpenTraceFromRealTimeLogger](#)

Opens an ETW trace processing handle for consuming events from an ETW real-time trace session or an ETW log file.

## [OpenTraceFromRealTimeLoggerWithAllocationOptions](#)

Creates a trace processing session attached to an active real-time ETW session.

## [OpenTraceW](#)

The OpenTraceW (Unicode) function (evntrace.h) opens an ETW trace processing handle for consuming events from an ETW real-time trace session or an ETW log file.

## [PEI\\_PROVIDER\\_NAME](#)

Macro that retrieves the Provider Event Info (PEI) name.

## [PENABLECALLBACK](#)

ETW event providers optionally define an EnableCallback function to receive configuration change notifications. The PENABLECALLBACK type defines a pointer to this callback function.

EnableCallback is a placeholder for the application-defined function name.

## [PETW\\_BUFFER\\_CALLBACK](#)

Function definition for the BufferCallback that will be invoked by ProcessTrace.

## [PETW\\_BUFFER\\_COMPLETION\\_CALLBACK](#)

Function definition for the callback that will be fired when ProcessTraceAddBufferToBufferStream is finished with a buffer. This callback should typically free the buffer as appropriate

## [PEVENT\\_CALLBACK](#)

ETW event consumers implement this callback to receive events from a trace processing session. The EventRecordCallback callback supersedes this callback.

## [PEVENT\\_RECORD\\_CALLBACK](#)

ETW event consumers implement this callback to receive events from a trace processing session. The PEVENT\_RECORD\_CALLBACK type defines a pointer to this callback function. EventRecordCallback is a placeholder for the application-defined function name.

## [PEVENT\\_TRACE\\_BUFFER\\_CALLBACKA](#)

The PEVENT\_TRACE\_BUFFER\_CALLBACKA (ANSI) (evntrace.h) function gets statistics about each buffer of events that ETW sends during a trace processing session.

## [PEVENT\\_TRACE\\_BUFFER\\_CALLBACKW](#)

The PEVENT\_TRACE\_BUFFER\_CALLBACKW (Unicode) (evntrace.h) function gets statistics about each buffer of events that ETW sends during a trace processing session.

## [PFI\\_FIELD\\_MESSAGE](#)

Macro that retrieves the Provider Field Information (PFI) field message.

## [PFI\\_FIELD\\_NAME](#)

Macro that retrieves the Provider Field Information (PFI) field name.

## [PFI\\_FILTER\\_MESSAGE](#)

Macro that filters the Provider Field Information (PFI) field message.

## [PFI\\_PROPERTY\\_NAME](#)

Macro that retrieves the Provider Field Information (PFI) property name.

## [ProcessTrace](#)

Delivers events from one or more trace processing sessions to the consumer.

## [ProcessTrace](#)

Delivers events from the associated trace streams to the consumer.

## [ProcessTraceAddBufferToBufferStream](#)

Provides an ETW trace buffer to a processing session created by OpenTraceFromBufferStream.

## [ProcessTraceBufferDecrementReference](#)

Releases a reference to a Buffer that was added by ProcessTraceBufferIncrementReference.

## [ProcessTraceBufferIncrementReference](#)

Called during the BufferCallback on the provided Buffer to prevent it from being freed until the caller is done with it.

## [QueryAllTracesA](#)

The QueryAllTracesA (ANSI) function (evntrace.h) function retrieves the properties and statistics for all event tracing sessions that the caller can query.

## [QueryAllTracesW](#)

The QueryAllTracesW (Unicode) function (evntrace.h) function retrieves the properties and statistics for all event tracing sessions that the caller can query.

## [QueryTraceA](#)

The QueryTraceA (ANSI) function (evntrace.h) retrieves the property settings and session statistics for the specified event tracing session.

## [QueryTraceProcessingHandle](#)

Retrieves information about an ETW trace processing session opened by OpenTrace.

## [QueryTraceW](#)

The QueryTraceW (Unicode) function (evntrace.h) retrieves the property settings and session statistics for the specified event tracing session.

## [RegisterCallback](#)

Registers an implementation of IEventCallback with the relogger in order to signal trace activity (starting, stopping, and logging new events).

## [RegisterTraceGuidsA](#)

The RegisterTraceGuidsA (ANSI) function (evntrace.h) is an obsolete function, and new code should use the provided alternative.

## [RegisterTraceGuidsW](#)

The RegisterTraceGuidsW (Unicode) function (evntrace.h) is an obsolete function, and new code should use the provided alternative.

## [RemoveTraceCallback](#)

The RemoveTraceCallback function stops an EventCallback function from receiving events for an event trace class. This function is obsolete.

## [SetActivityId](#)

Sets the activity ID in the current thread.

## [SetCompressionMode](#)

Enables or disables compression on the relogged trace.

## [SetEventDescriptor](#)

Sets the event descriptor for an event.

## [SetOutputFilename](#)

Indicates the file to which ETW should write the new, relogged trace.

## [SetPayload](#)

Sets the payload for an event.

## [SetProcessId](#)

Assigns an event to a specific process.

## [SetProcessorIndex](#)

Sets the processor index in the current thread.

## [SetProviderId](#)

Sets the GUID for the provider which traced an event.

## [SetThreadId](#)

Sets the identifier of a thread that generates an event.

## [SetThreadTimes](#)

Sets the thread times in the current thread.

## [SetTimeStamp](#)

Sets the time at which an event occurred.

## [SetTraceCallback](#)

The SetTraceCallback function specifies an EventCallback function to process events for the specified event trace class. This function is obsolete.

## [StartTraceA](#)

The StartTrace function starts an event tracing session. (ANSI)

## [StartTraceW](#)

The StartTrace function starts an event tracing session. (Unicode)

## [StopTraceA](#)

The StopTraceA (ANSI) function (`evntrace.h`) stops the specified event tracing session. The `ControlTrace` function supersedes this function.

## [StopTraceW](#)

The StopTraceW (Unicode) function (`evntrace.h`) stops the specified event tracing session. The `ControlTrace` function supersedes this function.

## [TdhAggregatePayloadFilters](#)

Aggregates multiple payload filters for a single provider into a single data structure for use with the `EnableTraceEx2` function.

## [TdhCleanupPayloadEventFilterDescriptor](#)

Frees the aggregated structure of payload filters created using the `TdhAggregatePayloadFilters` function.

## [TdhCloseDecodingHandle](#)

Frees any resources associated with the input decoding handle.

## [TdhCreatePayloadFilter](#)

Creates a single filter for a single payload to be used with the `EnableTraceEx2` function.

## [TdhDeletePayloadFilter](#)

Frees the memory allocated for a single payload filter by the `TdhCreatePayloadFilter` function.

## [TdhEnumerateManifestProviderEvents](#)

Retrieves the list of events present in the provider manifest.

## [TdhEnumerateProviderFieldInformation](#)

Retrieves the specified field metadata for a given provider.

## [TdhEnumerateProviderFilters](#)

Enumerates the filters that the specified provider defined in the manifest.

## [TdhEnumerateProviders](#)

Retrieves a list of providers that have registered a MOF class or manifest file on the computer.

[TdhEnumerateProvidersForDecodingSource](#)

Retrieves a list of providers that have registered a MOF class or manifest file on the computer.

[TdhFormatProperty](#)

Formats a property value for display.

[TdhGetDecodingParameter](#)

Retrieves the value of a decoding parameter.

[TdhGetEventInformation](#)

Retrieves metadata about an event.

[TdhGetEventMapInformation](#)

Retrieves information about the event map contained in the event.

[TdhGetManifestEventInformation](#)

Retrieves metadata about an event in a manifest.

[TdhGetProperty](#)

Retrieves a property value from the event data.

[TdhGetPropertySize](#)

Retrieves the size of one or more property values in the event data.

[TdhGetWppMessage](#)

Retrieves the formatted WPP message embedded into an EVENT\_RECORD structure.

[TdhGetWppProperty](#)

Retrieves a specific property associated with a WPP message.

[TdhLoadManifest](#)

Loads the manifest used to decode a log file.

[TdhLoadManifestFromBinary](#)

Takes a NULL-terminated path to a binary file that contains metadata resources needed to decode a specific event provider.

[TdhLoadManifestFromMemory](#)

Loads the manifest from memory.

[TdhOpenDecodingHandle](#)

Opens a decoding handle.

[TdhQueryProviderFieldInformation](#)

Retrieves information for the specified field from the event descriptions for those field values that match the given value.

[TdhSetDecodingParameter](#)

Sets the value of a decoding parameter.

[TdhUnloadManifest](#)

Unloads the manifest that was loaded by the TdhLoadManifest function.

[TdhUnloadManifestFromMemory](#)

Unloads the manifest from memory.

[TEI\\_ACTIVITYID\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) activity ID name.

[TEI\\_CHANNEL\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) channel name.

[TEI\\_EVENT\\_MESSAGE](#)

Macro that retrieves the Trace Event Information (TEI) message.

[TEI\\_KEYWORDS\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) keywords name.

[TEI\\_LEVEL\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) level name.

[TEI\\_MAP\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) map name.

## [TEI\\_OPCODE\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) opcode name.

## [TEI\\_PROPERTY\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) property name.

## [TEI\\_PROVIDER\\_MESSAGE](#)

Macro that retrieves the Trace Event Information (TEI) provider message.

## [TEI\\_PROVIDER\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) provider name.

## [TEI\\_RELATEDACTIVITYID\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) related activity ID name.

## [TEI\\_TASK\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) task name.

## [TraceEvent](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceEvent function to send a structured event to an event tracing session.

## [TraceEventInstance](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceEventInstance function to send a structured event to an event tracing session with an instance identifier.

## [TraceMessage](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceMessage function to send a message-based (TMF-based WPP) event to an event tracing session.

## [TraceMessageVa](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceMessageVa function to send a message-based (TMF-based WPP) event to an event tracing session using va\_list parameters.

## [TraceQueryInformation](#)

Provides information about an event tracing session.

|  |
|--|
|  |
| <a href="#">TraceSetInformation</a>  |
| Configures event tracing session settings.   |
| <a href="#">UnregisterTraceGuids</a>   |
| Unregisters a "Classic" (Windows 2000-style) ETW event trace provider that was registered using RegisterTraceGuids.  |
| <a href="#">UpdateTraceA</a>   |
| The UpdateTraceA (ANSI) function (evntrace.h) updates the property setting of the specified event tracing session.   |
| <a href="#">UpdateTraceW</a>   |
| The UpdateTraceW (Unicode) function (evntrace.h) updates the property setting of the specified event tracing session.  |
| <a href="#">WMIDPREQUEST</a>   |
| A RegisterTraceGuids-based ("Classic") event provider implements this function to receive notifications from controllers. The WMIDPREQUEST type defines a pointer to this callback function. ControlCallback is a placeholder for the application-defined function name. |

## Interfaces

|   |
|---|
|   |
| <a href="#">ITraceEvent</a>   |
| Provides access to data relating to a specific event.   |
| <a href="#">ITraceEventCallback</a>   |
| Used by ETW to provide information to the relogger as the tracing process starts, ends, and logs events.              |
| <a href="#">ITraceRelogger</a>  |
| Provides access to the relogging functionality, allowing you to manipulate and relog events from an ETW trace stream. |

## Structures

## [CLASSIC\\_EVENT\\_ID](#)

Identifies the kernel event for which you want to enable call stack tracing.

## [ENABLE\\_TRACE\\_PARAMETERS](#)

Contains information used to enable a provider via EnableTraceEx2.

## [ENABLE\\_TRACE\\_PARAMETERS\\_V1](#)

Contains information used to enable a provider via EnableTraceEx2. This structure is obsolete.

## [ETW\\_BUFFER\\_CALLBACK\\_INFORMATION](#)

Provided to the BufferCallback as the *ConsumerInfo* parameter and provides details on the current processing session.

## [ETW\\_BUFFER\\_CONTEXT](#)

Provides context information about the event.

## [ETW\\_BUFFER\\_CONTEXT](#)

Provides context information about the event. (ETW\_BUFFER\_CONTEXT)

## [ETW\\_BUFFER\\_HEADER](#)

The header structure of an ETW buffer.

## [ETW\\_OPEN\\_TRACE\\_OPTIONS](#)

Provides configuration parameters to OpenTraceFromBufferStream, OpenTraceFromFile, OpenTraceFromRealTimeLogger, OpenTraceFromRealTimeLoggerWithAllocationOptions functions.

## [ETW\\_TRACE\\_PARTITION\\_INFORMATION](#)

Contains partition information pulled from an ETW trace.

## [EVENT\\_DATA\\_DESCRIPTOR](#)

The EVENT\_DATA\_DESCRIPTOR structure defines a block of data that will be used in an ETW event.

## [EVENT\\_DESCRIPTOR](#)

The EVENT\_DESCRIPTOR structure contains information (metadata) about an ETW event.

## [EVENT\\_DESCRIPTOR](#)

Contains metadata that defines the event.

|  |
|--|
| EVENT_EXTENDED_ITEM_EVENT_KEY  |
| EVENT_EXTENDED_ITEM_INSTANCE   |
| Defines the relationship between events if TraceEventInstance was used to log related events.        |
| EVENT_EXTENDED_ITEM_PEBS_INDEX   |
| EVENT_EXTENDED_ITEM_PMC_COUNTERS   |
| EVENT_EXTENDED_ITEM_PROCESS_START_KEY  |
| EVENT_EXTENDED_ITEM RELATED_ACTIVITYID   |
| Defines the parent event of this event.  |
| EVENT_EXTENDED_ITEM_STACK_KEY32  |
| EVENT_EXTENDED_ITEM_STACK_KEY64  |
| EVENT_EXTENDED_ITEM_STACK_TRACE32  |
| Defines a call stack on a 32-bit computer.   |
| EVENT_EXTENDED_ITEM_STACK_TRACE64  |
| Defines a call stack on a 64-bit computer.   |
| EVENT_EXTENDED_ITEM_TS_ID  |
| Defines the terminal session that logged the event.  |
| EVENT_FILTER_DESCRIPTOR  |
| Defines the filter data that a session passes to the provider's enable callback function.            |
| EVENT_FILTER_EVENT_ID  |
| Defines event IDs used in an EVENT_FILTER_DESCRIPTOR structure for an event ID or stack walk filter. |

## [EVENT\\_FILTER\\_EVENT\\_NAME](#)

Defines event IDs used in an EVENT\_FILTER\_DESCRIPTOR structure for an event name or stalk walk name filter.

## [EVENT\\_FILTER\\_HEADER](#)

Defines the header data that must precede the filter data that is defined in the instrumentation manifest.

## [EVENT\\_FILTER\\_LEVEL\\_KW](#)

Defines event IDs used in an EVENT\_FILTER\_DESCRIPTOR structure for a stack walk level-keyword filter.

## [EVENT\\_HEADER](#)

The EVENT\_HEADER structure (evntcons.h) defines information about the event.

## [EVENT\\_HEADER](#)

The EVENT\_HEADER structure (relogger.h) defines information about the event.

## [EVENT\\_HEADER\\_EXTENDED\\_DATA\\_ITEM](#)

The EVENT\_HEADER\_EXTENDED\_DATA\_ITEM structure (evntcons.h) defines the extended data that ETW collects as part of the event data.

## [EVENT\\_HEADER\\_EXTENDED\\_DATA\\_ITEM](#)

The EVENT\_HEADER\_EXTENDED\_DATA\_ITEM structure (relogger.h) defines the extended data that ETW collects as part of the event data.

## [EVENT\\_INSTANCE\\_HEADER](#)

The EVENT\_INSTANCE\_HEADER structure contains standard event tracing information common to all events written by TraceEventInstance.

## [EVENT\\_INSTANCE\\_INFO](#)

The EVENT\_INSTANCE\_INFO structure maps a unique transaction identifier to a registered event trace class for TraceEventInstance.

## [EVENT\\_MAP\\_ENTRY](#)

Defines a single value map entry.

## [EVENT\\_MAP\\_INFO](#)

Defines the metadata about the event map.

## [EVENT\\_PROPERTY\\_INFO](#)

Provides information about a single property of the event or filter.

## [EVENT\\_RECORD](#)

The EVENT\_RECORD structure (`evntcons.h`) defines the layout of an event that ETW delivers.

## [EVENT\\_RECORD](#)

The EVENT\_RECORD structure (`relogger.h`) defines the layout of an event that ETW delivers.

## [EVENT\\_TRACE](#)

The EVENT\_TRACE structure is used to deliver event information to an event trace consumer.

## [EVENT\\_TRACE\\_HEADER](#)

The EVENT\_TRACE\_HEADER structure contains standard event tracing information common to all events written by `TraceEvent`.

## [EVENT\\_TRACE\\_LOGFILEA](#)

The EVENT\_TRACE\_LOGFILEA (ANSI) structure (`evntrace.h`) stores information about a trace data source.

## [EVENT\\_TRACE\\_LOGFILEW](#)

The EVENT\_TRACE\_LOGFILEW (Unicode) structure (`evntrace.h`) stores information about a trace data source.

## [EVENT\\_TRACE\\_PROPERTIES](#)

The EVENT\_TRACE\_PROPERTIES structure contains information about an event tracing session and is used with APIs such as `StartTrace` and `ControlTrace`.

## [EVENT\\_TRACE\\_PROPERTIES\\_V2](#)

The EVENT\_TRACE\_PROPERTIES\_V2 structure contains information about an event tracing session and is used with APIs such as `StartTrace` and `ControlTrace`.

## [MOF\\_FIELD](#)

You may use the MOF\_FIELD structures to append event data to the EVENT\_TRACE\_HEADER or EVENT\_INSTANCE\_HEADER structures.

|   |  |
|---|--|
| <a href="#">PAYLOAD_FILTER_PREDICATE</a>  | Defines an event payload filter predicate that describes how to filter on a single field in a trace session.         |
| <a href="#">PROPERTY_DATA_DESCRIPTOR</a>  | Defines the property to retrieve.  |
| <a href="#">PROVIDER_ENUMERATION_INFO</a> | Defines the array of providers that have registered a MOF or manifest on the computer.                               |
| <a href="#">PROVIDER_EVENT_INFO</a>       | Defines an array of events in a provider manifest.   |
| <a href="#">PROVIDER_FIELD_INFO</a>       | Defines the field information.   |
| <a href="#">PROVIDER_FIELD_INFOARRAY</a>  | Defines metadata information about the requested field.  |
| <a href="#">PROVIDER_FILTER_INFO</a>      | Defines a filter and its data.   |
| <a href="#">TDH_CONTEXT</a>               | Defines the additional information required to parse an event.   |
| <a href="#">TRACE_ENABLE_INFO</a>         | Defines the session and the information that the session used to enable the provider.                                |
| <a href="#">TRACE_EVENT_INFO</a>          | Defines the information about the event.   |
| <a href="#">TRACE_GUID_INFO</a>           | Returned by <code>EnumerateTraceGuidsEx</code> . Defines the header to the list of sessions that enabled a provider. |
| <a href="#">TRACE_GUID_PROPERTIES</a>     | Returned by <code>EnumerateTraceGuids</code> . Contains information about an event trace provider.                   |

## [TRACE\\_GUID\\_REGISTRATION](#)

Used with RegisterTraceGuids to register event trace classes.

## [TRACE\\_LOGFILE\\_HEADER](#)

The TRACE\_LOGFILE\_HEADER structure contains information about an event tracing session and its events.

## [TRACE\\_PERIODIC\\_CAPTURE\\_STATE\\_INFO](#)

Used with TraceQueryInformation and TraceSetInformation to get or set information relating to a periodic capture state.

## [TRACE\\_PROVIDER\\_INFO](#)

Defines the GUID and name for a provider.

## [TRACE\\_PROVIDER\\_INSTANCE\\_INFO](#)

Defines an instance of the provider GUID.

## [TRACE\\_VERSION\\_INFO](#)

Determines the version information of the TraceLogging session.

# EtwEventUnregister function

Article • 01/28/2023 • 2 minutes to read

## Description

Unregisters an event.

## Syntax

```
ULONG EVNTAPI EtwEventUnregister(
    [in] REGHANDLE RegHandle
);
```

## Parameters

### RegHandle [in]

Handle to an event.

## Returns

Returns an HRESULT.

## Remarks

Providers can only call this function from their [ControlCallback](#) function.

## See also

# evntcons.h header

Article • 01/24/2023 2 minutes to read

This header is used by Event Tracing. For more information, see:

- [Event Tracing](#)

evntcons.h contains the following programming interfaces:

## Functions

|  |
|--|
| <a href="#">EtwGetTraitFromProviderTraits</a>  |
| <a href="#">EventAccessControl</a>   |
| Adds or modifies the permissions of the specified provider or session.                 |
| <a href="#">EventAccessQuery</a>   |
| Retrieves the permissions for the specified controller or provider.                    |
| <a href="#">EventAccessRemove</a>  |
| Removes the permissions defined in the registry for the specified provider or session. |
| <a href="#">GetEventProcessorIndex</a>   |

## Structures

|   |
|---|
| <a href="#">EVENT_EXTENDED_ITEM_EVENT_KEY</a>   |
| <a href="#">EVENT_EXTENDED_ITEM_INSTANCE</a>  |
| Defines the relationship between events if TraceEventInstance was used to log related events. |
| <a href="#">EVENT_EXTENDED_ITEM_PEPS_INDEX</a>  |

[EVENT\\_EXTENDED\\_ITEM\\_PMC\\_COUNTERS](#)

[EVENT\\_EXTENDED\\_ITEM\\_PROCESS\\_START\\_KEY](#)

[EVENT\\_EXTENDED\\_ITEM RELATED\\_ACTIVITYID](#)

Defines the parent event of this event.

[EVENT\\_EXTENDED\\_ITEM\\_STACK\\_KEY32](#)

[EVENT\\_EXTENDED\\_ITEM\\_STACK\\_KEY64](#)

[EVENT\\_EXTENDED\\_ITEM\\_STACK\\_TRACE32](#)

Defines a call stack on a 32-bit computer.

[EVENT\\_EXTENDED\\_ITEM\\_STACK\\_TRACE64](#)

Defines a call stack on a 64-bit computer.

[EVENT\\_EXTENDED\\_ITEM\\_TS\\_ID](#)

Defines the terminal session that logged the event.

[EVENT\\_HEADER](#)

The EVENT\_HEADER structure (evntcons.h) defines information about the event.

[EVENT\\_HEADER\\_EXTENDED\\_DATA\\_ITEM](#)

The EVENT\_HEADER\_EXTENDED\_DATA\_ITEM structure (evntcons.h) defines the extended data that ETW collects as part of the event data.

[EVENT\\_RECORD](#)

The EVENT\_RECORD structure (evntcons.h) defines the layout of an event that ETW delivers.

## Enumerations

## [ETW\\_PROVIDER\\_TRAIT\\_TYPE](#)

Specifies the types of Provider Traits supported by Event Tracing for Windows (ETW).

## [EVENTSECURITYOPERATION](#)

Defines what component of the security descriptor that the EventAccessControl function modifies.

# evntprov.h header

Article • 01/24/2023 2 minutes to read

This header is used by multiple technologies. For more information, see:

- [Driver Development Tools Reference](#)
- [Event Tracing](#)

evntprov.h contains the following programming interfaces:

## Functions

|  |
|--|
| <a href="#">EventActivityIdControl</a>                                 |
| Creates, queries, and sets activity identifiers for use in ETW events. |
| <a href="#">EventDataDescCreate</a>                                    |
| Sets the values of an EVENT_DATA_DESCRIPTOR.                           |
| <a href="#">EventDescCreate</a>  |
| Sets the values of an event descriptor.                                |
| <a href="#">EventDescGetChannel</a>                                    |
| Retrieves the channel from the event descriptor.                       |
| <a href="#">EventDescGetId</a>   |
| Retrieves the event identifier from the event descriptor.              |
| <a href="#">EventDescGetKeyword</a>                                    |
| Retrieves the keyword from the event descriptor.                       |
| <a href="#">EventDescGetLevel</a>                                      |
| Retrieves the severity level from the event descriptor.                |
| <a href="#">EventDescGetOpcode</a>                                     |
| Retrieves the operation code from the event descriptor.                |

|  |
|--|
|  |
| <a href="#">EventDescGetTask</a>   |
| Retrieves the task from the event descriptor.  |
| <a href="#">EventDescGetVersion</a>  |
| Retrieves the version from the event descriptor.   |
| <a href="#">EventDescOrKeyword</a>   |
| Adds another keyword to the event descriptor.  |
| <a href="#">EventDescSetChannel</a>  |
| Sets the Channel member of the event descriptor.   |
| <a href="#">EventDescSetId</a>   |
| Sets the Id member of the event descriptor.  |
| <a href="#">EventDescSetKeyword</a>  |
| Sets the Keyword member of the event descriptor.   |
| <a href="#">EventDescSetLevel</a>  |
| Sets the Level member of the event descriptor.   |
| <a href="#">EventDescSetOpcode</a>   |
| Sets the Opcode member of the event descriptor.  |
| <a href="#">EventDescSetTask</a>   |
| Sets the Task member of the event descriptor.  |
| <a href="#">EventDescSetVersion</a>  |
| Sets the Version member of the event descriptor.   |
| <a href="#">EventDescZero</a>  |
| Initializes an event descriptor to zero.   |
| <a href="#">EventEnabled</a>   |
| Determines whether an event provider should generate a particular event based on the event's EVENT_DESCRIPTOR. |

|                                      |   |
|--------------------------------------|---|
| <a href="#">EventProviderEnabled</a> | Determines whether an event provider should generate a particular event based on the event's Level and Keyword. |
| <a href="#">EventRegister</a>        | Registers an ETW event provider, creating a handle that can be used to write ETW events.                        |
| <a href="#">EventSetInformation</a>  | Configures an ETW event provider.   |
| <a href="#">EventUnregister</a>      | Unregisters an ETW event provider.  |
| <a href="#">EventWrite</a>           | Writes an ETW event that uses the current thread's activity ID.   |
| <a href="#">EventWriteEx</a>         | Writes an ETW event with an activity ID, an optional related activity ID, session filters, and special options. |
| <a href="#">EventWriteString</a>     | Writes an ETW event that contains a string as its data. This function should not be used.                       |
| <a href="#">EventWriteTransfer</a>   | Writes an ETW event with an activity ID and an optional related activity ID.                                    |

## Callback functions

|                                 |  |
|---------------------------------|--|
| <a href="#">PENABLECALLBACK</a> | ETW event providers optionally define an EnableCallback function to receive configuration change notifications. The PENABLECALLBACK type defines a pointer to this callback function. EnableCallback is a placeholder for the application-defined function name. |
|---------------------------------|--|

## Structures

## [EVENT\\_DATA\\_DESCRIPTOR](#)

The EVENT\_DATA\_DESCRIPTOR structure defines a block of data that will be used in an ETW event.

## [EVENT\\_DESCRIPTOR](#)

The EVENT\_DESCRIPTOR structure contains information (metadata) about an ETW event.

## [EVENT\\_FILTER\\_DESCRIPTOR](#)

Defines the filter data that a session passes to the provider's enable callback function.

## [EVENT\\_FILTER\\_EVENT\\_ID](#)

Defines event IDs used in an EVENT\_FILTER\_DESCRIPTOR structure for an event ID or stack walk filter.

## [EVENT\\_FILTER\\_EVENT\\_NAME](#)

Defines event IDs used in an EVENT\_FILTER\_DESCRIPTOR structure for an event name or stalk walk name filter.

## [EVENT\\_FILTER\\_HEADER](#)

Defines the header data that must precede the filter data that is defined in the instrumentation manifest.

## [EVENT\\_FILTER\\_LEVEL\\_KW](#)

Defines event IDs used in an EVENT\_FILTER\_DESCRIPTOR structure for a stack walk level-keyword filter.

# Enumerations

## [EVENT\\_INFO\\_CLASS](#)

The EVENT\_INFO\_CLASS enumeration type is used with the EventSetInformation function to specify the configuration operation to be performed on an ETW event provider registration.

# evntrace.h header

Article • 01/24/2023 7 minutes to read

This header is used by multiple technologies. For more information, see:

- [Event Tracing](#)
- [Kernel-Mode Driver Reference](#)

evntrace.h contains the following programming interfaces:

## Functions

|  |
|--|
| <a href="#">CloseTrace</a>   |
| The CloseTrace function closes a trace processing session that was created with OpenTrace.   |
| <a href="#">ControlTraceA</a>  |
| The ControlTraceA (ANSI) function (evntrace.h) flushes, queries, updates, or stops the specified event tracing session.  |
| <a href="#">ControlTraceW</a>  |
| The ControlTraceW (Unicode) function (evntrace.h) flushes, queries, updates, or stops the specified event tracing session.   |
| <a href="#">CreateTracelInstanceld</a>   |
| A RegisterTraceGuids-based ("Classic") event provider uses the CreateTracelInstanceld function to create a unique transaction identifier and map it to a registration handle. The provider can then use the transaction identifier when calling the TraceEventInstance function. |
| <a href="#">EnableTrace</a>  |
| A trace session controller calls EnableTrace to configure how an ETW event provider logs events to a trace session. The EnableTraceEx2 function supersedes this function.  |
| <a href="#">EnableTraceEx</a>  |
| A trace session controller calls EnableTraceEx to configure how an ETW event provider logs events to a trace session. The EnableTraceEx2 function supersedes this function.  |

## [EnableTraceEx2](#)

A trace session controller calls EnableTraceEx2 to configure how an ETW event provider logs events to a trace session.

## [EnumerateTraceGuids](#)

Retrieves information about event trace providers that are currently running on the computer. The EnumerateTraceGuidsEx function supersedes this function.

## [EnumerateTraceGuidsEx](#)

Retrieves information about event trace providers that are currently running on the computer.

## [FlushTraceA](#)

The FlushTraceA (ANSI) function (evntrace.h) causes an event tracing session to immediately deliver buffered events for the specified session.

## [FlushTraceW](#)

The FlushTraceW (Unicode) function (evntrace.h) causes an event tracing session to immediately deliver buffered events for the specified session.

## [GetTraceEnableFlags](#)

A RegisterTraceGuids-based ("Classic") event provider uses the GetTraceEnableFlags function to retrieve the enable flags specified by the trace controller to indicate which category of events to trace. Providers call this function from their ControlCallback function.

## [GetTraceEnableLevel](#)

A RegisterTraceGuids-based ("Classic") event provider uses the GetTraceEnableLevel function to retrieve the enable level specified by the trace controller to indicate which level of events to trace. Providers call this function from their ControlCallback function.

## [GetTraceLoggerHandle](#)

A RegisterTraceGuids-based ("Classic") event provider uses the GetTraceLoggerHandle function to retrieve the handle of the event tracing session to which it should write events. Providers call this function from their ControlCallback function.

## [OpenTraceA](#)

The OpenTraceA (ANSI) function (evntrace.h) opens an ETW trace processing handle for consuming events from an ETW real-time trace session or an ETW log file.

|  |
|--|
|  |
| <p><a href="#">OpenTraceFromBufferStream</a></p> <p>Creates a trace processing session that is not directly attached to any file or active session.</p>  |
| <p><a href="#">OpenTraceFromFile</a></p> <p>Creates a trace processing session to process a Tracelog .etl file.</p>  |
| <p><a href="#">OpenTraceFromRealTimeLogger</a></p> <p>Opens an ETW trace processing handle for consuming events from an ETW real-time trace session or an ETW log file.</p>                                  |
| <p><a href="#">OpenTraceFromRealTimeLoggerWithAllocationOptions</a></p> <p>Creates a trace processing session attached to an active real-time ETW session.</p>   |
| <p><a href="#">OpenTraceW</a></p> <p>The OpenTraceW (Unicode) function (evntrace.h) opens an ETW trace processing handle for consuming events from an ETW real-time trace session or an ETW log file.</p>    |
| <p><a href="#">ProcessTrace</a></p> <p>Delivers events from one or more trace processing sessions to the consumer.</p>   |
| <p><a href="#">ProcessTraceAddBufferToBufferStream</a></p> <p>Provides an ETW trace buffer to a processing session created by OpenTraceFromBufferStream.</p>   |
| <p><a href="#">ProcessTraceBufferDecrementReference</a></p> <p>Releases a reference to a Buffer that was added by ProcessTraceBufferIncrementReference.</p>  |
| <p><a href="#">ProcessTraceBufferIncrementReference</a></p> <p>Called during the BufferCallback on the provided Buffer to prevent it from being freed until the caller is done with it.</p>                  |
| <p><a href="#">QueryAllTracesA</a></p> <p>The QueryAllTracesA (ANSI) function (evntrace.h) function retrieves the properties and statistics for all event tracing sessions that the caller can query.</p>    |
| <p><a href="#">QueryAllTracesW</a></p> <p>The QueryAllTracesW (Unicode) function (evntrace.h) function retrieves the properties and statistics for all event tracing sessions that the caller can query.</p> |

## [QueryTraceA](#)

The QueryTraceA (ANSI) function (evntrace.h) retrieves the property settings and session statistics for the specified event tracing session.

## [QueryTraceProcessingHandle](#)

Retrieves information about an ETW trace processing session opened by OpenTrace.

## [QueryTraceW](#)

The QueryTraceW (Unicode) function (evntrace.h) retrieves the property settings and session statistics for the specified event tracing session.

## [RegisterTraceGuidsA](#)

The RegisterTraceGuidsA (ANSI) function (evntrace.h) is an obsolete function, and new code should use the provided alternative.

## [RegisterTraceGuidsW](#)

The RegisterTraceGuidsW (Unicode) function (evntrace.h) is an obsolete function, and new code should use the provided alternative.

## [RemoveTraceCallback](#)

The RemoveTraceCallback function stops an EventCallback function from receiving events for an event trace class. This function is obsolete.

## [SetTraceCallback](#)

The SetTraceCallback function specifies an EventCallback function to process events for the specified event trace class. This function is obsolete.

## [StartTraceA](#)

The StartTrace function starts an event tracing session. (ANSI)

## [StartTraceW](#)

The StartTrace function starts an event tracing session. (Unicode)

## [StopTraceA](#)

The StopTraceA (ANSI) function (evntrace.h) stops the specified event tracing session. The ControlTrace function supersedes this function.

## [StopTraceW](#)

The StopTraceW (Unicode) function (evntrace.h) stops the specified event tracing session. The ControlTrace function supersedes this function.

## [TraceEvent](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceEvent function to send a structured event to an event tracing session.

## [TraceEventInstance](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceEventInstance function to send a structured event to an event tracing session with an instance identifier.

## [TraceMessage](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceMessage function to send a message-based (TMF-based WPP) event to an event tracing session.

## [TraceMessageVa](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceMessageVa function to send a message-based (TMF-based WPP) event to an event tracing session using va\_list parameters.

## [TraceQueryInformation](#)

Provides information about an event tracing session.

## [TraceSetInformation](#)

Configures event tracing session settings.

## [UnregisterTraceGuids](#)

Unregisters a "Classic" (Windows 2000-style) ETW event trace provider that was registered using RegisterTraceGuids.

## [UpdateTraceA](#)

The UpdateTraceA (ANSI) function (evntrace.h) updates the property setting of the specified event tracing session.

## [UpdateTraceW](#)

The UpdateTraceW (Unicode) function (evntrace.h) updates the property setting of the specified event tracing session.

# Callback functions

## PETW\_BUFFER\_CALLBACK

Function definition for the BufferCallback that will be invoked by ProcessTrace.

## PETW\_BUFFER\_COMPLETION\_CALLBACK

Function definition for the callback that will be fired when ProcessTraceAddBufferToBufferStream is finished with a buffer. This callback should typically free the buffer as appropriate

## PEVENT\_CALLBACK

ETW event consumers implement this callback to receive events from a trace processing session. The EventRecordCallback callback supersedes this callback.

## PEVENT\_RECORD\_CALLBACK

ETW event consumers implement this callback to receive events from a trace processing session. The PEVENT\_RECORD\_CALLBACK type defines a pointer to this callback function. EventRecordCallback is a placeholder for the application-defined function name.

## PEVENT\_TRACE\_BUFFER\_CALLBACKA

The PEVENT\_TRACE\_BUFFER\_CALLBACKA (ANSI) (evntrace.h) function gets statistics about each buffer of events that ETW sends during a trace processing session.

## PEVENT\_TRACE\_BUFFER\_CALLBACKW

The PEVENT\_TRACE\_BUFFER\_CALLBACKW (Unicode) (evntrace.h) function gets statistics about each buffer of events that ETW sends during a trace processing session.

## WMIDPREQUEST

A RegisterTraceGuids-based ("Classic") event provider implements this function to receive notifications from controllers. The WMIDPREQUEST type defines a pointer to this callback function. ControlCallback is a placeholder for the application-defined function name.

# Structures

## CLASSIC\_EVENT\_ID

Identifies the kernel event for which you want to enable call stack tracing.

|   |  |
|---|--|
| <a href="#">ENABLE_TRACE_PARAMETERS</a>         | Contains information used to enable a provider via <code>EnableTraceEx2</code> .   |
| <a href="#">ENABLE_TRACE_PARAMETERS_V1</a>      | Contains information used to enable a provider via <code>EnableTraceEx2</code> . This structure is obsolete.   |
| <a href="#">ETW_BUFFER_CALLBACK_INFORMATION</a> | Provided to the <code>BufferCallback</code> as the <code>ConsumerInfo</code> parameter and provides details on the current processing session.   |
| <a href="#">ETW_BUFFER_CONTEXT</a>              | Provides context information about the event.  |
| <a href="#">ETW_BUFFER_HEADER</a>               | The header structure of an ETW buffer.   |
| <a href="#">ETW_OPEN_TRACE_OPTIONS</a>          | Provides configuration parameters to <code>OpenTraceFromBufferStream</code> , <code>OpenTraceFromFile</code> , <code>OpenTraceFromRealTimeLogger</code> , <code>OpenTraceFromRealTimeLoggerWithAllocationOptions</code> functions. |
| <a href="#">ETW_TRACE_PARTITION_INFORMATION</a> | Contains partition information pulled from an ETW trace.   |
| <a href="#">EVENT_INSTANCE_HEADER</a>           | The <code>EVENT_INSTANCE_HEADER</code> structure contains standard event tracing information common to all events written by <code>TraceEventInstance</code> .   |
| <a href="#">EVENT_INSTANCE_INFO</a>             | The <code>EVENT_INSTANCE_INFO</code> structure maps a unique transaction identifier to a registered event trace class for <code>TraceEventInstance</code> .  |
| <a href="#">EVENT_TRACE</a>                     | The <code>EVENT_TRACE</code> structure is used to deliver event information to an event trace consumer.  |
| <a href="#">EVENT_TRACE_HEADER</a>              | The <code>EVENT_TRACE_HEADER</code> structure contains standard event tracing information common to all events written by <code>TraceEvent</code> .  |

## [EVENT\\_TRACE\\_LOGFILEA](#)

The EVENT\_TRACE\_LOGFILEA (ANSI) structure (evntrace.h) stores information about a trace data source.

## [EVENT\\_TRACE\\_LOGFILEW](#)

The EVENT\_TRACE\_LOGFILEW (Unicode) structure (evntrace.h) stores information about a trace data source.

## [EVENT\\_TRACE\\_PROPERTIES](#)

The EVENT\_TRACE\_PROPERTIES structure contains information about an event tracing session and is used with APIs such as StartTrace and ControlTrace.

## [EVENT\\_TRACE\\_PROPERTIES\\_V2](#)

The EVENT\_TRACE\_PROPERTIES\_V2 structure contains information about an event tracing session and is used with APIs such as StartTrace and ControlTrace.

## [MOF\\_FIELD](#)

You may use the MOF\_FIELD structures to append event data to the EVENT\_TRACE\_HEADER or EVENT\_INSTANCE\_HEADER structures.

## [TRACE\\_ENABLE\\_INFO](#)

Defines the session and the information that the session used to enable the provider.

## [TRACE\\_GUID\\_INFO](#)

Returned by EnumerateTraceGuidsEx. Defines the header to the list of sessions that enabled a provider.

## [TRACE\\_GUID\\_PROPERTIES](#)

Returned by EnumerateTraceGuids. Contains information about an event trace provider.

## [TRACE\\_GUID\\_REGISTRATION](#)

Used with RegisterTraceGuids to register event trace classes.

## [TRACE\\_LOGFILE\\_HEADER](#)

The TRACE\_LOGFILE\_HEADER structure contains information about an event tracing session and its events.

### [TRACE\\_PERIODIC\\_CAPTURE\\_STATE\\_INFO](#)

Used with TraceQueryInformation and TraceSetInformation to get or set information relating to a periodic capture state.

### [TRACE\\_PROVIDER\\_INSTANCE\\_INFO](#)

Defines an instance of the provider GUID.

### [TRACE\\_VERSION\\_INFO](#)

Determines the version information of the TraceLogging session.

## Enumerations

### [ETW\\_PROCESS\\_HANDLE\\_INFO\\_TYPE](#)

Specifies the operation that will be performed on a trace processing session.

### [ETW\\_PROCESS\\_TRACE\\_MODES](#)

Specifies the supported process trace modes.

### [TRACE\\_QUERY\\_INFO\\_CLASS](#)

Used with EnumerateTraceGuidsEx and TraceSetInformation to specify a type of trace information.

# relogger.h header

Article • 01/24/2023 2 minutes to read

This header is used by Event Tracing. For more information, see:

- [Event Tracing](#)

relogger.h contains the following programming interfaces:

## Interfaces

### [ITraceEvent](#)

Provides access to data relating to a specific event.

### [ITraceEventCallback](#)

Used by ETW to provide information to the relogger as the tracing process starts, ends, and logs events.

### [ITraceRelogger](#)

Provides access to the relogging functionality, allowing you to manipulate and relog events from an ETW trace stream.

## Functions

### [DECLSPEC\\_XFGVIRT](#)

## Structures

### [ETW\\_BUFFER\\_CONTEXT](#)

Provides context information about the event. (ETW\_BUFFER\_CONTEXT)

## [EVENT\\_DESCRIPTOR](#)

Contains metadata that defines the event.

## [EVENT\\_HEADER](#)

The EVENT\_HEADER structure (relogger.h) defines information about the event.

## [EVENT\\_HEADER\\_EXTENDED\\_DATA\\_ITEM](#)

The EVENT\_HEADER\_EXTENDED\_DATA\_ITEM structure (relogger.h) defines the extended data that ETW collects as part of the event data.

## [EVENT\\_RECORD](#)

The EVENT\_RECORD structure (relogger.h) defines the layout of an event that ETW delivers.

# CveEventWrite function (securitybaseapi.h)

Article • 12/09/2022 2 minutes to read

A tracing function for publishing events when an attempted security vulnerability exploit is detected in your user-mode application.

## Syntax

C++

```
LONG CveEventWrite(
    [in]          PCWSTR CveId,
    [in, optional] PCWSTR AdditionalDetails
);
```

## Parameters

[in] CveId

A pointer to the CVE ID associated with the vulnerability for which this event is being raised.

[in, optional] AdditionalDetails

A pointer to a string giving additional details that the event producer may want to provide to the consumer of this event.

## Return value

Returns ERROR\_SUCCESS if successful or one of the following values on error.

| Return code               | Description   |
|---------------------------|---|
| ERROR_INVALID_PARAMETER   | One or more of the parameters is not valid.                       |
| ERROR_ARITHMETIC_OVERFLOW | The event size is larger than the allowed maximum (64k - header). |
| ERROR_MORE_DATA           | The session buffer size is too small for the event.               |
| ERROR_NOT_ENOUGH_MEMORY   | Occurs when filled buffers are trying to flush to disk, but       |

disk IOs are not happening fast enough. This happens when the disk is slow and event traffic is heavy.

Eventually, there are no more free (empty) buffers and the event is dropped.

#### STATUS\_LOG\_FILE\_FULL

The real-time playback file is full. Events are not logged to the session until a real-time consumer consumes the events from the playback file. Do not stop logging events based on this error code.

## Remarks

The CveEventWrite function publishes a CVE-based event. This function should be called only in scenarios where an attempt to exploit a known, patched vulnerability is detected by the application. Ideally, this function call should be added as part of the fix (update) itself.

The default consumer for this event is EventLog-Application. To enable another consumer, the provider can be added to the consumer session.

Provider GUID: 85a62a0d-7e17-485f-9d4f-749a287193a6

Source Name: Microsoft-Windows-Audit-CVE or Audit-CVE

## Requirements

|                          |   |
|--------------------------|---|
| Minimum supported client | Windows 10 [desktop apps   UWP apps]          |
| Minimum supported server | Windows Server 2016 [desktop apps   UWP apps] |
| Target Platform          | Windows                                       |
| Header                   | securitybaseapi.h (include Windows.h)         |
| Library                  | Advapi32.lib                                  |
| DLL                      | Advapi32.dll                                  |

# tdh.h header

Article • 01/24/2024 4 minutes to read

This trace data helper (TDH) header is used by Event Tracing. For more information, see:

- [Event Tracing](#)

tdh.h contains the following programming interfaces:

## Functions

|  |
|--|
|  |
| <a href="#">EMI_MAP_FORMAT</a>   |
| Macro that retrieves the event map format.                               |
| <a href="#">EMI_MAP_INPUT</a>  |
| Macro that retrieves the event map input.                                |
| <a href="#">EMI_MAP_NAME</a>   |
| Macro that retrieves the event map name.                                 |
| <a href="#">EMI_MAP_OUTPUT</a>   |
| Macro that retrieves the event map output.                               |
| <a href="#">PEI_PROVIDER_NAME</a>  |
| Macro that retrieves the Provider Event Info (PEI) name.                 |
| <a href="#">PFI_FIELD_MESSAGE</a>  |
| Macro that retrieves the Provider Field Information (PFI) field message. |
| <a href="#">PFI_FIELD_NAME</a>   |
| Macro that retrieves the Provider Field Information (PFI) field name.    |
| <a href="#">PFI_FILTER_MESSAGE</a>                                       |
| Macro that filters the Provider Field Information (PFI) field message.   |

|  |  |
|--|--|
| PFI_PROPERTY_NAME                      | Macro that retrieves the Provider Field Information (PFI) property name.   |
| TdhAggregatePayloadFilters             | Aggregates multiple payload filters for a single provider into a single data structure for use with the EnableTraceEx2 function. |
| TdhCleanupPayloadEventFilterDescriptor | Frees the aggregated structure of payload filters created using the TdhAggregatePayloadFilters function.                         |
| TdhCloseDecodingHandle                 | Frees any resources associated with the input decoding handle.   |
| TdhCreatePayloadFilter                 | Creates a single filter for a single payload to be used with the EnableTraceEx2 function.  |
| TdhDeletePayloadFilter                 | Frees the memory allocated for a single payload filter by the TdhCreatePayloadFilter function.                                   |
| TdhEnumerateManifestProviderEvents     | Retrieves the list of events present in the provider manifest.   |
| TdhEnumerateProviderFieldInformation   | Retrieves the specified field metadata for a given provider.   |
| TdhEnumerateProviderFilters            | Enumerates the filters that the specified provider defined in the manifest.  |
| TdhEnumerateProviders                  | Retrieves a list of providers that have registered a MOF class or manifest file on the computer.                                 |
| TdhEnumerateProvidersForDecodingSource | Retrieves a list of providers that have registered a MOF class or manifest file on the computer.                                 |
| TdhFormatProperty                      | Formats a property value for display.  |

[TdhGetDecodingParameter](#)

Retrieves the value of a decoding parameter.

[TdhGetEventInformation](#)

Retrieves metadata about an event.

[TdhGetEventMapInformation](#)

Retrieves information about the event map contained in the event.

[TdhGetManifestEventInformation](#)

Retrieves metadata about an event in a manifest.

[TdhGetProperty](#)

Retrieves a property value from the event data.

[TdhGetPropertySize](#)

Retrieves the size of one or more property values in the event data.

[TdhGetWppMessage](#)

Retrieves the formatted WPP message embedded into an EVENT\_RECORD structure.

[TdhGetWppProperty](#)

Retrieves a specific property associated with a WPP message.

[TdhLoadManifest](#)

Loads the manifest used to decode a log file.

[TdhLoadManifestFromBinary](#)

Takes a NULL-terminated path to a binary file that contains metadata resources needed to decode a specific event provider.

[TdhLoadManifestFromMemory](#)

Loads the manifest from memory.

[TdhOpenDecodingHandle](#)

Opens a decoding handle.

|  |
|--|
|  |
| <a href="#">TdhQueryProviderFieldInformation</a>   |
| Retrieves information for the specified field from the event descriptions for those field values that match the given value. |
| <a href="#">TdhSetDecodingParameter</a>  |
| Sets the value of a decoding parameter.  |
| <a href="#">TdhUnloadManifest</a>  |
| Unloads the manifest that was loaded by the TdhLoadManifest function.  |
| <a href="#">TdhUnloadManifestFromMemory</a>  |
| Unloads the manifest from memory.  |
| <a href="#">TEI_ACTIVITYID_NAME</a>  |
| Macro that retrieves the Trace Event Information (TEI) activity ID name.   |
| <a href="#">TEI_CHANNEL_NAME</a>   |
| Macro that retrieves the Trace Event Information (TEI) channel name.   |
| <a href="#">TEI_EVENT_MESSAGE</a>  |
| Macro that retrieves the Trace Event Information (TEI) message.  |
| <a href="#">TEI_KEYWORDS_NAME</a>  |
| Macro that retrieves the Trace Event Information (TEI) keywords name.  |
| <a href="#">TEI_LEVEL_NAME</a>   |
| Macro that retrieves the Trace Event Information (TEI) level name.   |
| <a href="#">TEI_MAP_NAME</a>   |
| Macro that retrieves the Trace Event Information (TEI) map name.   |
| <a href="#">TEI_OPCODE_NAME</a>  |
| Macro that retrieves the Trace Event Information (TEI) opcode name.  |
| <a href="#">TEI_PROPERTY_NAME</a>  |
| Macro that retrieves the Trace Event Information (TEI) property name.  |

## [TEI\\_PROVIDER\\_MESSAGE](#)

Macro that retrieves the Trace Event Information (TEI) provider message.

## [TEI\\_PROVIDER\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) provider name.

## [TEI RELATEDACTIVITYID\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) related activity ID name.

## [TEI\\_TASK\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) task name.

# Structures

## [EVENT\\_MAP\\_ENTRY](#)

Defines a single value map entry.

## [EVENT\\_MAP\\_INFO](#)

Defines the metadata about the event map.

## [EVENT\\_PROPERTY\\_INFO](#)

Provides information about a single property of the event or filter.

## [PAYLOAD\\_FILTER\\_PREDICATE](#)

Defines an event payload filter predicate that describes how to filter on a single field in a trace session.

## [PROPERTY\\_DATA\\_DESCRIPTOR](#)

Defines the property to retrieve.

## [PROVIDER\\_ENUMERATION\\_INFO](#)

Defines the array of providers that have registered a MOF or manifest on the computer.

## [PROVIDER\\_EVENT\\_INFO](#)

Defines an array of events in a provider manifest.

## [PROVIDER\\_FIELD\\_INFO](#)

Defines the field information.

## [PROVIDER\\_FIELD\\_INFOARRAY](#)

Defines metadata information about the requested field.

## [PROVIDER\\_FILTER\\_INFO](#)

Defines a filter and its data.

## [TDH\\_CONTEXT](#)

Defines the additional information required to parse an event.

## [TRACE\\_EVENT\\_INFO](#)

Defines the information about the event.

## [TRACE\\_PROVIDER\\_INFO](#)

Defines the GUID and name for a provider.

# Enumerations

## [\\_TDH\\_IN\\_TYPE](#)

Defines the supported [in] types for a trace data helper (TDH).

## [\\_TDH\\_OUT\\_TYPE](#)

Defines the supported [out] types for a trace data helper (TDH).

## [DECODING\\_SOURCE](#)

Defines the source of the event data.

## [EVENT\\_FIELD\\_TYPE](#)

Defines the provider information to retrieve.

## [MAP\\_FLAGS](#)

Defines constant values that indicate if the map is a value map, bitmap, or pattern map.

## [MAP\\_VALUETYPE](#)

Defines if the value map value is in a ULONG data type or a string.

## [PAYLOAD\\_OPERATOR](#)

Defines the supported payload operators for a trace data helper (TDH).

## [PROPERTY\\_FLAGS](#)

Defines if the property is contained in a structure or array.

## [TDH\\_CONTEXT\\_TYPE](#)

Defines the context type.

## [TEMPLATE\\_FLAGS](#)

Defines constant values that indicates the layout of the event data.

# TdhGetAllEventsInformation function

Article • 03/21/2023 • 2 minutes to read

Retrieves metadata about all events in the specified namespace.

## Syntax

C++

```
TdhGetAllEventsInformation(
    __in PEVENT_RECORD pEvent,
    __in_opt PVOID pWbemService,
    __out ULONG *pIndex,
    __out ULONG *pCount,
    __out_bcount_opt(*pBufferSize) PTRACE_EVENT_INFO *ppBuffer,
    __inout ULONG *pBufferSize
);
```

## Parameters

### pEvent [in]

The event record passed to your [EventRecordCallback](#) callback. For details, see the [EVENT\\_RECORD](#) structure.

### pWbemService [in]

Pointer to namespace for which the diagnostic mode data is retrieved.

### pIndex [out]

The index in the *ppBuffer* buffer of the first **PTRACE\_EVENT\_INFO**.

### pCount

The number of **PTRACE\_EVENT\_INFO** entries returned in *ppBuffer*.

### ppBuffer

User-allocated buffer to receive the event information. For details, see the [TRACE\\_EVENT\\_INFO](#) structure.

## pBufferSize

Size, in bytes, of the *pBuffer* buffer. If the function succeeds, this parameter receives the size of the buffer used. If the buffer is too small, the function returns `ERROR_INSUFFICIENT_BUFFER` and sets this parameter to the required buffer size. If the buffer size is zero on input, no data is returned in the buffer and this parameter receives the required buffer size.

## Return value

`S_OK` on success.

## Remarks

This function is not defined in an SDK header and must be declared by the caller. This function is exported from `tdh.dll`.

## Requirements

| Requirement              | Value                |
|--------------------------|----------------------|
| Minimum supported client | Windows 11           |
| Minimum supported server | Windows 11           |
| DLL                      | <code>tdh.dll</code> |

## See also

# Event Tracing MOF Classes

Article • 01/07/2021 • 2 minutes to read

The classes are defined in the \root\wmi namespace. The **EventVersion** qualifier for the kernel events associates the class with a specific operating system. For example, version zero is associated with Windows 2000, version one with Windows XP and Windows Server 2003, and version two with Windows Vista.

The following MOF classes define the ETW event classes.

- [EventTrace](#)
- [EventTrace\\_Header](#)
- [EventTraceEvent](#)
- [Lost\\_Event](#)
- [MSNT\\_SystemTrace](#)
- [RT\\_LostEvent](#)

The following MOF classes define the kernel events classes.

- [ALPC](#)
- [ALPC\\_Receive\\_Message](#)
- [ALPC\\_Send\\_Message](#)
- [ALPC\\_Unwait](#)
- [ALPC\\_Wait\\_For\\_New\\_Message](#)
- [ALPC\\_Wait\\_For\\_Reply](#)
- [CSwitch](#)
- [DPC](#)
- [DiskIo](#)
- [DiskIo\\_TypeGroup1](#)
- [DiskIo\\_TypeGroup2](#)
- [DiskIo\\_TypeGroup3](#)
- [DriverCompleteRequest](#)
- [DriverCompleteRequestReturn](#)
- [DriverCompletionRoutine](#)
- [DriverMajorFunctionCall](#)
- [DriverMajorFunctionReturn](#)
- [FileIo](#)
- [FileIo\\_Create](#)
- [FileIo\\_DirEnum](#)
- [FileIo\\_Info](#)
- [FileIo\\_Name](#)

- FileIo\_OpEnd
- FileIo\_ReadWrite
- FileIo\_SimpleOp
- FileIo\_V0
- FileIo\_V0\_Name
- FileIo\_V1
- FileIo\_V1\_Name
- HWConfig
- HWConfig\_CPU
- HWConfig\_LogDisk
- HWConfig\_NIC
- HWConfig\_PhysDisk
- Image
- Image\_Load
- Image\_V0
- Image\_V0\_Load
- Image\_V1
- Image\_V1\_Load
- ISR
- PageFault\_V2
- PageFault\_HardFault
- PageFault\_ImageLoadBacked
- PageFault\_TypeGroup1
- PerfInfo
- Process
- Process\_TypeGroup1
- Process\_V0
- Process\_V0\_TypeGroup1
- Process\_V1
- Process\_V1\_TypeGroup1
- Process\_V2
- Process\_V2\_TypeGroup1
- Process\_V2\_TypeGroup2
- ReadyThread
- Registry
- Registry\_TypeGroup1
- Registry\_V0
- Registry\_V0\_TypeGroup1
- Registry\_V1
- Registry\_V1\_TypeGroup1

- [SampledProfile](#)
- [SplitIo](#)
- [SplitIo\\_Info](#)
- [SysCallEnter](#)
- [SysCallExit](#)
- [SystemConfig](#)
- [SystemConfig\\_CPU](#)
- [SystemConfig\\_IDEChannel](#)
- [SystemConfig\\_IRQ](#)
- [SystemConfig\\_LogDisk](#)
- [SystemConfig\\_Network](#)
- [SystemConfig\\_NIC](#)
- [SystemConfig\\_PhysDisk](#)
- [SystemConfig\\_PnP](#)
- [SystemConfig\\_Power](#)
- [SystemConfig\\_Services](#)
- [SystemConfig\\_Video](#)
- [SystemConfig\\_V0\\_CPU](#)
- [SystemConfig\\_V0\\_LogDisk](#)
- [SystemConfig\\_V0\\_NIC](#)
- [SystemConfig\\_V0\\_PhysDisk](#)
- [SystemConfig\\_V0\\_Power](#)
- [SystemConfig\\_V0\\_Services](#)
- [SystemConfig\\_V0\\_Video](#)
- [Tcplp](#)
- [Tcplp\\_Fail](#)
- [Tcplp\\_SendIPV4](#)
- [Tcplp\\_SendIPV6](#)
- [Tcplp\\_TypeGroup1](#)
- [Tcplp\\_TypeGroup2](#)
- [Tcplp\\_TypeGroup3](#)
- [Tcplp\\_TypeGroup4](#)
- [Tcplp\\_V0](#)
- [Tcplp\\_V0\\_TypeGroup1](#)
- [Tcplp\\_V1](#)
- [Tcplp\\_V1\\_TypeGroup1](#)
- [Thread](#)
- [Thread\\_TypeGroup1](#)
- [Thread\\_V0](#)
- [Thread\\_V0\\_TypeGroup1](#)

- Thread\_V1
- Thread\_V1\_TypeGroup1
- Thread\_V1\_TypeGroup2
- Thread\_V2
- Thread\_V2\_TypeGroup1
- Udplp
- Udplp\_Fail
- Udplp\_TypeGroup1
- Udplp\_TypeGroup2
- Udplp\_V0
- Udplp\_V0\_TypeGroup1
- Udplp\_V1
- Udplp\_V1\_TypeGroup1

# EventTrace class

Article • 01/07/2021 • 2 minutes to read

An abstract class from which all event trace classes are derived.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[abstract]
class EventTrace
{
    uint16 EventSize;
    uint16 ReservedHeaderField;
    uint8 EventType;
    uint8 TraceLevel;
    uint16 TraceVersion;
    uint64 ThreadId;
    uint64 TimeStamp;
    uint8 EventGuid[];
    uint32 KernelTime;
    uint32 UserTime;
    uint32 InstanceId;
    uint8 ParentGuid[];
    uint32 ParentInstanceId;
    uint32 MofData;
    uint32 MofLength;
};
```

## Members

The **EventTrace** class has these types of members:

- [Properties](#)

## Properties

The **EventTrace** class has these properties.

### EventGuid

Data type: **uint8** array

Access type: Read-only

Qualifiers: **WmiDataId** (8), **Max** (16)

Event trace class GUID of this event.

### **EventSize**

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Total number of bytes of the event.

### **EventType**

Data type: **uint8**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Provider-defined event type. Tells you which event type class to use to decipher the provider-defined event data (the data pointed to by **MofData**).

### **InstanceId**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (11)

Identifier of this event instance.

### **KernelTime**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (9)

Elapsed execution time for kernel-mode instructions, in CPU ticks.

### **MofData**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (14), **Pointer**

Pointer to the provider-specific event data.

### **MofLength**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (15)

Length of the provider-specific event data.

### **ParentGuid**

Data type: **uint8 array**

Access type: Read-only

Qualifiers: **WmiDataId** (12), **Max** (16)

Event trace class GUID of the parent instance.

### **ParentInstanceId**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (13)

Identifier of the parent instance data.

### **ReservedHeaderField**

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Reserved.

### **ThreadId**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (6), **Pointer**

Identifies the thread that generated the event.

### **TimeStamp**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (7)

Contains the date and time when the event occurred.

### **TraceLevel**

Data type: **uint8**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Provider-defined value that defines the severity level used to generate the event.

### **TraceVersion**

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Provider-defined version number of the event trace class used to generate the event.

### **UserTime**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (10)

Elapsed execution time for user-mode instructions, in CPU ticks.

## Remarks

Do not use these properties.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |
| MOF                      | Wmi.mof                                       |

# MSNT\_SystemTrace class

Article • 01/07/2021 • 2 minutes to read

The parent class from which all system event trace classes are derived.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{9e814aad-3204-11d2-9a82-006008a86939}")]
class MSNT_SystemTrace : EventTrace
{
    uint32 Flags;
};
```

## Members

The **MSNT\_SystemTrace** class has these types of members:

- [Properties](#)

## Properties

The **MSNT\_SystemTrace** class has these properties.

### Flags

Data type: **uint32**

Access type: Read-only

Qualifiers: `DefineValues{"EVENT_TRACE_FLAG_PROCESS", "EVENT_TRACE_FLAG_THREAD", "EVENT_TRACE_FLAG_IMAGE_LOAD", "EVENT_TRACE_FLAG_PROCESS_COUNTERS", "EVENT_TRACE_FLAG_CSWITCH", "EVENT_TRACE_FLAG_DPC", "EVENT_TRACE_FLAG_INTERRUPT", "EVENT_TRACE_FLAG_SYSTEMCALL", "EVENT_TRACE_FLAG_DISK_IO", "EVENT_TRACE_FLAG_DISK_FILE_IO", "EVENT_TRACE_FLAG_DISK_IO_INIT", "EVENT_TRACE_FLAG_DISPATCHER", "EVENT_TRACE_FLAG_MEMORY_PAGEFAULTS", "EVENT_TRACE_FLAG_MEMORY_HARDFAULTS", "EVENT_TRACE_FLAG_VIRTUAL_ALLOC", "EVENT_TRACE_FLAG_NETWORK_TCPIP", }`

```
"EVENT_TRACE_FLAG_REGISTRY", "EVENT_TRACE_FLAG_ALPC",
"EVENT_TRACE_FLAG_SPLIT_IO", "EVENT_TRACE_FLAG_DRIVER",
"EVENT_TRACE_FLAG_PROFILE", "EVENT_TRACE_FLAG_FILE_IO",
"EVENT_TRACE_FLAG_FILE_IO_INIT"}, ValueMap{"0x00000001", "0x00000002",
"0x00000004", "0x00000008", "0x00000010", "0x00000020", "0x00000040",
"0x00000080", "0x00000100", "0x00000200", "0x00000400", "0x00000800",
"0x00001000", "0x00002000", "0x00004000", "0x00010000", "0x00020000",
"0x00100000", "0x00200000", "0x00800000", "0x01000000", "0x02000000",
"0x04000000"}
```

Enable flag that indicates the enabled kernel providers.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

# ALPC class

Article • 01/07/2021 • 2 minutes to read

This class is the parent class for advanced local procedure call events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{45d8cccd-539f-4b72-a8b7-5c683142609a}")]
class ALPC : MSNT_SystemTrace
{
};
```

## Members

The ALPC class does not define any members.

## Remarks

To enable advanced local procedure call events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_ALPC** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function.

Event trace consumers can implement special processing for ALPC events by calling the [SetTraceCallback](#) function and specifying **ALPCGuid** as the *pGuid* parameter. Use the following event types to identify the actual ALPC event when consuming events.

| Event type           | Description  |
|----------------------|--|
| Event type value, 33 | Send message event. The <a href="#">ALPC_Send_Message</a> MOF class defines the event data for this event.       |
| Event type value, 34 | Receive message event. The <a href="#">ALPC_Receive_Message</a> MOF class defines the event data for this event. |
| Event type value, 35 | Wait for reply event. The <a href="#">ALPC_Wait_For_Reply</a> MOF class defines the event data for this event.   |

| <b>Event type</b>    | <b>Description</b>   |
|----------------------|--|
| Event type value, 36 | Wait for new message event. The <a href="#">ALPC_Wait_For_New_Message</a> MOF class defines the event data for this event. |
| Event type value, 37 | Stop waiting event. The <a href="#">ALPC_Unwait</a> MOF class defines the event data for this event.                       |

## Requirements

| <b>Requirement</b>       | <b>Value</b>                            |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# ALPC\_Receive\_Message class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for ALPC receive message events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(34), EventTypeName("ALPC-Receive-Message")]
class ALPC_Receive_Message : ALPC
{
    uint32 MessageID;
};
```

## Members

The **ALPC\_Receive\_Message** class has these types of members:

- [Properties](#)

## Properties

The **ALPC\_Receive\_Message** class has these properties.

### MessageID

Data type: **uint32**

Access type: Read-only

Message identifier.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# ALPC\_Send\_Message class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for ALPC send message events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(33), EventTypeName("ALPC-Send-Message")]
class ALPC_Send_Message : ALPC
{
    uint32 MessageID;
};
```

## Members

The **ALPC\_Send\_Message** class has these types of members:

- [Properties](#)

## Properties

The **ALPC\_Send\_Message** class has these properties.

### MessageID

Data type: **uint32**

Access type: Read-only

Message identifier.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# ALPC\_Unwait class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for ALPC end wait events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(37), EventTypeName("ALPC-Unwait")]
class ALPC_Unwait : ALPC
{
    uint32 Status;
};
```

## Members

The **ALPC\_Unwait** class has these types of members:

- [Properties](#)

## Properties

The **ALPC\_Unwait** class has these properties.

### Status

Data type: **uint32**

Access type: Read-only

Status from wait operation.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# ALPC\_Wait\_For\_New\_Message class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for ALPC wait for new message events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(36), EventTypeName("ALPC-Wait-For-New-Message")]
class ALPC_Wait_For_New_Message : ALPC
{
    uint32 IsServerPort;
    string PortName;
};
```

## Members

The **ALPC\_Wait\_For\_New\_Message** class has these types of members:

- [Properties](#)

## Properties

The **ALPC\_Wait\_For\_New\_Message** class has these properties.

### IsServerPort

Data type: **uint32**

Access type: Read-only

Indicates if the port is a server port.

### PortName

Data type: **string**

Access type: Read-only

String that contains the name of the port on which ALPC is waiting.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# ALPC\_Wait\_For\_Reply class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for ALPC wait for reply events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(35), EventTypeName("ALPC-Wait-For-Reply")]
class ALPC_Wait_For_Reply : ALPC
{
    uint32 MessageID;
};
```

## Members

The **ALPC\_Wait\_For\_Reply** class has these types of members:

- [Properties](#)

## Properties

The **ALPC\_Wait\_For\_Reply** class has these properties.

### MessageID

Data type: **uint32**

Access type: Read-only

Message identifier.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# DiskIo class

Article • 01/07/2021 • 2 minutes to read

This class is the parent class for disk I/O events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{3d6fa8d4-fe05-11d0-9dda-00c04fd7ba7c}")]
class DiskIo : MSNT_SystemTrace
{
};
```

## Members

The **DiskIo** class does not define any members.

## Remarks

To enable disk I/O events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_DISK\_IO** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function. You can also specify one or more of the following flags:

- **EVENT\_TRACE\_FLAG\_DISK\_IO\_INIT**
- **EVENT\_TRACE\_FLAG\_DRIVER**

Event trace consumers can implement special processing for disk I/O events by calling the [SetTraceCallback](#) function and specifying **DiskIoGuid** as the *pGuid* parameter. Use the following event types to identify the actual disk I/O event when consuming events.

| Event type  | Description   |
|---|---|
| <b>EVENT_TRACE_TYPE_IO_READ</b> (Event type value is 10)  | Read event. The <a href="#">DiskIo_TypeGroup1</a> MOF class defines the event data for this event.  |
| <b>EVENT_TRACE_TYPE_IO_WRITE</b> (Event type value is 11) | Write event. The <a href="#">DiskIo_TypeGroup1</a> MOF class defines the event data for this event. |

| Event type  | Description  |
|---|--|
| EVENT_TRACE_TYPE_IO_READ_INIT(Event type value is 12)       | Initialize read event. The <a href="#">DiskIo_TypeGroup2</a> MOF class defines the event data for this event.                                |
| EVENT_TRACE_TYPE_IO_WRITE_INIT(Event type value is 13)      | Initialize write event. The <a href="#">DiskIo_TypeGroup2</a> MOF class defines the event data for this event.                               |
| EVENT_TRACE_TYPE_IO_FLUSH(Event type value is 14)           | Initialize write event. The <a href="#">DiskIo_TypeGroup3</a> MOF class defines the event data for this event.                               |
| EVENT_TRACE_TYPE_IO_FLUSH_INIT(Event type value is 15)      | Initialize flush event. The <a href="#">DiskIo_TypeGroup2</a> MOF class defines the event data for this event.                               |
| EVENT_TRACE_TYPE_IO_REDIRECTED_INIT(Event type value is 16) | Initialize redirected event. Redirected IO events are used to map disk IOs to a Windows Imaging Format (WIM) to the filename within the WIM. |
| Event type value is 52                                      | Driver complete request event. The <a href="#">DriverCompleteRequest</a> MOF class defines the event data for this event.                    |
| Event type value is 53                                      | Driver complete request return event. The <a href="#">DriverCompleteRequestReturn</a> MOF class defines the event data for this event.       |
| Event type value is 37                                      | Driver completion routine event. The <a href="#">DriverCompletionRoutine</a> MOF class defines the event data for this event.                |
| Event type value is 34                                      | Driver major function call event. The <a href="#">DriverMajorFunctionCall</a> MOF class defines the event data for this event.               |
| Event type value is 35                                      | Driver major function call return event. The <a href="#">DriverMajorFunctionReturn</a> MOF class defines the event data for this event.      |

The disk I/O provider cannot identify which file is read or written during a disk I/O event. To retrieve the name of the file associated with the disk I/O event, enable the file I/O event provider.

Disk I/O events are recorded at the I/O completion time. To determine when the I/O operation began, use the initialization events, for example, EVENT\_TRACE\_TYPE\_IO\_READ\_INIT.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[DiskIo\\_TypeGroup1](#)

[DiskIo\\_TypeGroup2](#)

[DiskIo\\_TypeGroup3](#)

[DriverCompleteRequest](#)

[DriverCompleteRequestReturn](#)

[DriverCompletionRoutine](#)

[DriverMajorFunctionCall](#)

[DriverMajorFunctionReturn](#)

# DiskIo\_TypeGroup1 class

Article • 01/07/2021 • 3 minutes to read

This class is the event type class for disk I/O events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10,11}, EventTypeName{"Read","Write"}]
class DiskIo_TypeGroup1 : DiskIo
{
    uint32 DiskNumber;
    uint32 IrpFlags;
    uint32 TransferSize;
    uint32 Reserved;
    sint64 ByteOffset;
    uint32 FileObject;
    uint32 Irp;
    uint64 HighResResponseTime;
    uint32 IssuingThreadId;
};
```

## Members

The **DiskIo\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **DiskIo\_TypeGroup1** class has these properties.

### ByteOffset

Data type: **sint64**

Access type: Read-only

Qualifiers: [WmiDataId](#) (5)

Byte offset from the beginning of the physical disk.

## DiskNumber

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (1)

Number that identifies the physical disk.

## FileObject

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (6), [Pointer](#)

Match the value of this pointer to the **FileObject** pointer value in a [FileIo\\_Name](#) event to determine the file involved in the I/O operation.

## HighResResponseTime

Data type: **uint64**

Access type: Read-only

Qualifiers: [WmiDataId](#) (8)

The time between I/O initiation and completion as measured by the partition manager (in the [KeQueryPerformanceCounter](#) tick units).

**Windows Server 2003:** This property has a [WmiDataId](#) value of 7.

**Windows 2000 Server and Windows 2000 Professional:** This property is not supported.

## Irp

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (7), [Pointer](#)

The I/O request packet, which identifies the I/O activity.

**Windows Server 2003, Windows 2000 Server and Windows 2000 Professional:** This property is not supported.

## IrpFlags

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (2), [Format](#) ("x")

Can contain one or more of the following I/O request packet flags (defined in Ntddk.h, which is a DDK header file):

**IRP\_NOCACHE**

**IRP\_PAGING\_IO**

**IRP\_MOUNT\_COMPLETION**

**IRP\_SYNCHRONOUS\_API**

**IRP\_ASSOCIATED\_IRP**

**IRP\_BUFFERED\_IO**

**IRP\_DEALLOCATE\_BUFFER**

**IRP\_INPUT\_OPERATION**

**IRP\_SYNCHRONOUS\_PAGING\_IO**

**IRP\_CREATE\_OPERATION**

**IRP\_READ\_OPERATION**

**IRP\_WRITE\_OPERATION**

**IRP\_CLOSE\_OPERATION**

**IRP\_DEFER\_IO\_COMPLETION**

## IssuingThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (9)

The identifier of the issuing thread.

**Windows Server 2008 R2, Windows Server 2008, Windows 7, Windows Vista, Windows Server 2003 with SP1, Windows Server 2003, Windows 2000 Server and Windows 2000 Professional:** This property is not supported.

#### Reserved

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (4)

Reserved.

**Windows Server 2008 R2, Windows Server 2008 and Windows 7:** The name of the property is **QueueDepth**, which contains the CPU tick count from the beginning of the operation to the end of the operation. Note that this value can overflow.

**Windows Vista, Windows Server 2003 with SP1, Windows Server 2003, Windows 2000 Server and Windows 2000 Professional:** The name of the property is **ResponseTime**, which contains the CPU tick count from the beginning of the operation to the end of the operation. Note that this value can overflow.

#### TransferSize

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (3)

Size of data read to or written from disk, in bytes.

## Remarks

Windows Server 2003 uses the following definition for the **DiskIo\_TypeGroup1** event type class.

### syntax

```
[EventType{10, 11}, EventTypeName{"Read", "Write"}]
class DiskIo_TypeGroup1 : DiskIo
{
    [WmiDataId(1), read] uint32 DiskNumber;
    [WmiDataId(2), format("x"), read] uint32 IrpFlags;
    [WmiDataId(3), read] uint32 TransferSize;
    [WmiDataId(4), read] uint32 ResponseTime;
```

```

    [WmiDataId(5), read] uint64 ByteOffset;
    [WmiDataId(6), pointer, read] uint32 FileObject;
    [WmiDataId(7), read] uint64 HighResResponseTime;
};


```

The **ResponseTime** property contains the CPU tick count from the beginning of the operation to the end of the operation. Note that this value can overflow.

The **HighResResponseTime** property is not supported.

Windows Server 2003 with SP1 and Windows Vista uses the following definition for the **DiskIo\_TypeGroup1** event type class.

#### syntax

```

[EventType{10, 11}, EventTypeName{"Read", "Write"}]
class DiskIo_TypeGroup1 : DiskIo
{
    [WmiDataId(1), read] uint32 DiskNumber;
    [WmiDataId(2), format("x"), read] uint32 IrpFlags;
    [WmiDataId(3), read] uint32 TransferSize;
    [WmiDataId(4), read] uint32 ResponseTime;
    [WmiDataId(5), read] uint64 ByteOffset;
    [WmiDataId(6), pointer, read] uint32 FileObject;
    [WmiDataId(7), pointer, read] uint32 Irp;
    [WmiDataId(8), read] uint64 HighResResponseTime;
};


```

The **Irp** property is the I/O request packet. This property identifies the I/O activity. You can use this property with the [DiskIo\\_TypeGroup2](#) events to correlate response time.

The **HighResResponseTime** property is supported. The property contains the time between I/O initiation and completion as measured by PartitionManager (in the KeQueryPerformanceCounter units). Use this property instead of the **ResponseTime** property to determine the disk I/O response time.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

Disklo

# DiskIo\_TypeGroup2 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class that marks the beginning of the disk I/O read, write, and flush events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{12, 13, 15}, EventTypeName{"ReadInit", "WriteInit", "FlushInit"}]
class DiskIo_TypeGroup2 : DiskIo
{
    uint32 Irp;
    uint32 IssuingThreadId;
};
```

## Members

The **DiskIo\_TypeGroup2** class has these types of members:

- [Properties](#)

## Properties

The **DiskIo\_TypeGroup2** class has these properties.

### Irp

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (1), [Pointer](#)

I/O request packet. This property identifies the IO activity.

### IssuingThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (2)

The identifier of the issuing thread.

**Windows Server 2008 R2, Windows Server 2008, Windows 7 and Windows Vista:** This property is not supported.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Disklo](#)

# DiskIo\_TypeGroup3 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for disk I/O flush events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{14}, EventTypeName{"FlushBuffers"}]
class DiskIo_TypeGroup3 : DiskIo
{
    uint32 DiskNumber;
    uint32 IrpFlags;
    uint64 HighResResponseTime;
    uint32 Irp;
    uint32 IssuingThreadId;
};
```

## Members

The **DiskIo\_TypeGroup3** class has these types of members:

- [Properties](#)

## Properties

The **DiskIo\_TypeGroup3** class has these properties.

### DiskNumber

Data type: **uint32**

Access type: Read-only

Qualifiers: [WmiDataId](#) (1)

Number that identifies the physical disk.

### HighResResponseTime

Data type: **uint64**

Access type: Read-only

Qualifiers: [WmiDataId](#) (3)

CPU tick count from the beginning of the operation to the end of the operation.

## Irp

Data type: `uint32`

Access type: Read-only

Qualifiers: [WmiDataId](#) (4), [Pointer](#)

I/O request packet. This property identifies the I/O activity.

## IrpFlags

Data type: `uint32`

Access type: Read-only

Qualifiers: [WmiDataId](#) (2), [Format](#) ("x")

Can contain one or more of the following I/O request packet flags (defined in Ntddk.h, which is a DDK header file):

`IRP_NO_CACHE`

`IRP_PAGING_IO`

`IRP_MOUNT_COMPLETION`

`IRP_SYNCHRONOUS_API`

`IRP_ASSOCIATED_IRP`

`IRP_BUFFERED_IO`

`IRP_DEALLOCATE_BUFFER`

`IRP_INPUT_OPERATION`

`IRP_SYNCHRONOUS_PAGING_IO`

`IRP_CREATE_OPERATION`

`IRP_READ_OPERATION`

**IRP\_WRITE\_OPERATION**

**IRP\_CLOSE\_OPERATION**

**IRP\_DEFER\_IO\_COMPLETION**

### IssuingThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

The identifier of the issuing thread.

**Windows Server 2008 R2, Windows Server 2008, Windows 7 and Windows Vista:** This property is not supported.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Disklo](#)

# DriverCompleteRequest class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for driver complete request events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{52}, EventTypeName{"DrvComplReq"}]
class DriverCompleteRequest : DiskIo
{
    uint32 RoutineAddr;
    uint32 Irp;
    uint32 UniqMatchId;
};
```

## Members

The **DriverCompleteRequest** class has these types of members:

- [Properties](#)

## Properties

The **DriverCompleteRequest** class has these properties.

### Irp

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

IO request packet.

### RoutineAddr

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Address of the driver function being called.

### UniqMatchId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Identifier that uniquely identifies request. Use this identifier to correlate with the other driver events, for example, the [DriverCompleteRequestReturn](#) event.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[DiskIo](#)

# DriverCompleteRequestReturn class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for driver complete request return events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{53}, EventTypeName{"DrvComplReqRet"}]
class DriverCompleteRequestReturn : DiskIo
{
    uint32 Irp;
    uint32 UniqMatchId;
};
```

## Members

The **DriverCompleteRequestReturn** class has these types of members:

- [Properties](#)

## Properties

The **DriverCompleteRequestReturn** class has these properties.

### Irp

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

IO request packet.

### UniqMatchId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Identifier that uniquely identifies request. Use this identifier to correlate with the other driver events, for example, the [DriverCompleteRequest](#) event.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[DiskIo](#)

# DriverCompletionRoutine class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for driver complete routine events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{37}, EventTypeName{"DrvComplRout"}]
class DriverCompletionRoutine : DiskIo
{
    uint32 Routine;
    uint32 Irp;
    uint32 UniqMatchId;
};
```

## Members

The **DriverCompletionRoutine** class has these types of members:

- [Properties](#)

## Properties

The **DriverCompletionRoutine** class has these properties.

### Irp

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

IO request packet.

### Routine

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Address of the driver function being called.

### UniqMatchId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Identifier that uniquely identifies request. Use this identifier to correlate with the other driver events, for example, the [DriverCompleteRequest](#) event.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[DiskIo](#)

# DriverMajorFunctionCall class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for driver major function call events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{34}, EventTypeName{"DrvMjFnCall"}]
class DriverMajorFunctionCall : DiskIo
{
    uint32 MajorFunction;
    uint32 MinorFunction;
    uint32 RoutineAddr;
    uint32 FileObject;
    uint32 Irp;
    uint32 UniqMatchId;
};
```

## Members

The **DriverMajorFunctionCall** class has these types of members:

- [Properties](#)

## Properties

The **DriverMajorFunctionCall** class has these properties.

### FileObject

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(4)**, **Pointer**

Match the value of this pointer to the **FileObject** pointer value in a [DiskIo\\_TypeGroup1](#) event to determine the type of I/O operation.

### Irp

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5), Pointer

IO request packet.

### **MajorFunction**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Code that identifies the major function being called.

### **MinorFunction**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Code that identifies the minor function being called.

### **RoutineAddr**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Address of the driver function being called.

### **UniqMatchId**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6)

Identifier that uniquely identifies request. Use this identifier to correlate with the other driver events, for example, the [DriverCompleteRequest](#) event.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Disklo](#)

# DriverMajorFunctionReturn class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for driver major function call return events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{35}, EventTypeName{"DrvMjFnRet"}]
class DriverMajorFunctionReturn : DiskIo
{
    uint32 Irp;
    uint32 UniqMatchId;
};
```

## Members

The **DriverMajorFunctionReturn** class has these types of members:

- [Properties](#)

## Properties

The **DriverMajorFunctionReturn** class has these properties.

### Irp

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

IO request packet.

### UniqMatchId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Identifier that uniquely identifies request. Use this identifier to correlate with the other driver events, for example, the [DriverCompleteRequest](#) event.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[DiskIo](#)

# EventTraceEvent class

Article • 01/07/2021 • 2 minutes to read

The parent class for log header events. This class is always the first event trace class sent to a consumer (this event is not sent to real-time consumers).

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{68fdd900-4a3e-11d1-84f4-0000f80464e3}")]
class EventTraceEvent : MSNT_SystemTrace
{
};
```

## Members

The **EventTraceEvent** class does not define any members.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[EventTrace\\_Header](#)

# EventTrace\_Header class

Article • 01/07/2021 • 3 minutes to read

The event type class for the log file header event. This class contains information about the event tracing session.

The following syntax is simplified from MOF code.

## Syntax

```
syntax

[EventType(0)]
class EventTrace_Header : EventTraceEvent
{
    uint32 BufferSize;
    uint32 Version;
    uint32 ProviderVersion;
    uint32 NumberOfProcessors;
    uint64 EndTime;
    uint32 TimerResolution;
    uint32 MaxFileSize;
    uint32 FileMode;
    uint32 BuffersWritten;
    uint32 StartBuffers;
    uint32 PointerSize;
    uint32 EventsLost;
    uint32 CPUSpeed;
    uint32 LoggerName;
    uint32 LogFileName;
    uint8 TimeZoneInformation[];
    uint64 BootTime;
    uint64 PerfFreq;
    uint64 StartTime;
    uint32 ReservedFlags;
    uint32 BuffersLost;
};
```

## Members

The **EventTrace\_Header** class has these types of members:

- [Properties](#)

## Properties

The **EventTrace\_Header** class has these properties.

#### **BootTime**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (17)

Time at which the system was started, in 100-nanosecond intervals since midnight, January 1, 1601.

#### **BufferSize**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Size of the event tracing session's buffers, in kilobytes.

#### **BuffersLost**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (21)

Total number of buffers lost.

#### **BuffersWritten**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (9)

Total number of buffers written by the event tracing session.

#### **CPUSpeed**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (13)

CPU speed, in megahertz.

**Windows 2000:** Not supported.

### **EndTime**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Time at which the event tracing session stopped, in 100-nanosecond intervals since midnight, January 1, 1601. This value may be 0 if you are consuming events in real time or from a log file to which the provider is still logging events.

### **EventsLost**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (12)

Number of events lost during the event tracing session.

### **LogFileMode**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (8), **Format("x")**

Current logging mode for the event tracing session. For a list of values, see Logging Mode Constants.

### **LogFileName**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (15), **Pointer**

Name of the event tracing log file that contains the events.

## **LoggerName**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (14), **Pointer**

Name of the event tracing session.

## **MaxFileSize**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (7)

Maximum size of the log file, in megabytes.

## **NumberOfProcessors**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Number of processors on the system.

## **PerfFreq**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (18)

Frequency of the high-resolution performance counter, if one exists.

## **PointerSize**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (11)

Size of a pointer data type, in bytes.

## **ProviderVersion**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Build number of the operating system.

## **ReservedFlags**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (20)

Reserved.

## **StartBuffers**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (10)

Reserved.

## **StartTime**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (19)

Time at which the event tracing session started, in 100-nanosecond intervals since midnight, January 1, 1601.

## **TimerResolution**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (6)

Resolution of the hardware timer, in units of 100 nanoseconds.

### TimeZoneInformation

Data type: **uint8** array

Access type: Read-only

Qualifiers: **WmiDataId** (16), **Extension("NoPrint")**, **Max** (176)

A [TIME\\_ZONE\\_INFORMATION](#) structure that contains the time zone for the **BootTime**, **EndTime** and **StartTime** members.

### Version

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Version number of the operating system. Starting with the low-order bytes, the first two bytes contain major version, the next two bytes contain minor version, the next two bytes contain service pack major version, and the last two bytes contain service pack minor version.

## Remarks

Typically, you want to save the values for the following properties for use later when processing events from the log file.

- **TimerResolution**—use with the **KernelTime** and **UserTime** members of the [EVENT\\_TRACE\\_HEADER](#) structure to determine the CPU cost for a set of instructions. For details, see the Remarks section of [EVENT\\_TRACE\\_HEADER](#).
- **PointerSize**—for properties that contain the **Pointer** qualifier, use this value to determine the size of the pointer. Note that this value may not be accurate. For example, on a 64-bit computer, a 32-bit application will log 4-byte pointers; however, the session will set **PointerSize** to 8.
- **LogFileMode**—use to determine if this session is a private logger session. There are some properties that do not contain data for private logger sessions. For example, the **KernelTime** and **UserTime** members of the [EVENT\\_TRACE\\_HEADER](#) structure.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[EventTraceEvent](#)

[TRACE\\_LOGFILE\\_HEADER](#)

# FileIo class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for file I/O events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{90cbdc39-4a3e-11d1-84f4-0000f80464e3"}), EventVersion(2)]
class FileIo : MSNT_SystemTrace
{
};
```

## Members

The **FileIo** class does not define any members.

## Remarks

To enable the File IO events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_DISK\_FILE\_IO** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function. You can also specify one or more of the following flags:

- **EVENT\_TRACE\_FLAG\_FILE\_IO**
- **EVENT\_TRACE\_FLAG\_FILE\_IO\_INIT**

Event trace consumers can implement special processing for file I/O events by calling the [SetTraceCallback](#) function and specifying **FileIoGuid** as the *pGuid* parameter. Use the following event types to identify the actual event when consuming events.

| Event type            | Description   |
|-----------------------|---|
| Event type value is 0 | File name event. The <a href="#">FileIo_Name</a> MOF class defines the event data for this event. |

| <b>Event type</b>      | <b>Description</b>  |
|------------------------|---|
| Event type value is 32 | File create event. The <a href="#">Filelo_Name</a> MOF class defines the event data for this event.   |
| Event type value is 35 | File delete event. The <a href="#">Filelo_Name</a> MOF class defines the event data for this event.   |
| Event type value is 36 | File rundown event. Enumerates all open files on the computer at the end of the trace session. The <a href="#">Filelo_Name</a> MOF class defines the event data for this event. |
| Event type value is 64 | File create event. The <a href="#">Filelo_Create</a> MOF class defines the event data for this event.   |
| Event type value is 72 | Directory enumeration event. The <a href="#">Filelo_DirEnum</a> MOF class defines the event data for this event.  |
| Event type value is 77 | Directory notification event. The <a href="#">Filelo_DirEnum</a> MOF class defines the event data for this event.   |
| Event type value is 69 | Set information event. The <a href="#">Filelo_Info</a> MOF class defines the event data for this event.   |
| Event type value is 70 | Delete file event. The <a href="#">Filelo_Info</a> MOF class defines the event data for this event.   |
| Event type value is 71 | Rename file event. The <a href="#">Filelo_Info</a> MOF class defines the event data for this event.   |
| Event type value is 74 | Query file information event. The <a href="#">Filelo_Info</a> MOF class defines the event data for this event.  |
| Event type value is 75 | File system control event. The <a href="#">Filelo_Info</a> MOF class defines the event data for this event.   |
| Event type value is 76 | End of operation event. The <a href="#">Filelo_OpEnd</a> MOF class defines the event data for this event.   |

| <b>Event type</b>      | <b>Description</b>   |
|------------------------|--|
| Event type value is 67 | File read event. The <a href="#">FileIo_ReadWrite</a> MOF class defines the event data for this event.   |
| Event type value is 68 | File write event. The <a href="#">FileIo_ReadWrite</a> MOF class defines the event data for this event.  |
| Event type value is 65 | Clean up event. The event is generated when the last handle to the file is released. The <a href="#">FileIo_SimpleOp</a> MOF class defines the event data for this event.  |
| Event type value is 66 | Close event. The event is generated when the file object is freed. The <a href="#">FileIo_SimpleOp</a> MOF class defines the event data for this event.                    |
| Event type value is 73 | Flush event. This event is generated when the file buffers are fully flushed to disk. The <a href="#">FileIo_SimpleOp</a> MOF class defines the event data for this event. |

File IO events are logged at the beginning of the operation.

## Requirements

| <b>Requirement</b>       | <b>Value</b>                            |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[FileIo\\_V0](#)

[FileIo\\_V1](#)

# FileIo\_Create class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for the file create event.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{64}, EventTypeName{"Create"}]
class FileIo_Create : FileIo
{
    uint32 IrpPtr;
    uint32 TTID;
    uint32 FileObject;
    uint32 CreateOptions;
    uint32 FileAttributes;
    uint32 ShareAccess;
    string OpenPath;
};
```

## Members

The **FileIo\_Create** class has these types of members:

- [Properties](#)

## Properties

The **FileIo\_Create** class has these properties.

### CreateOptions

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(4)**

Values passed in the *CreateOptions* and *CreateDispositions* parameters to the [\*\*NtCreateFile\*\*](#) function.

## **FileAttributes**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5)

Value passed in the *FileAttributes* parameter to the [NtCreateFile](#) function.

## **FileObject**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Identifier that can be used for correlating operations to the same opened file object instance between file create and close events.

## **IrpPtr**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

IO request packet. This property identifies the IO activity.

## **OpenPath**

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(7), StringTermination("NullTerminated"), Format("w")

Path to the file.

## **ShareAccess**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6)

Value passed in the *ShareAccess* parameter to the [NtCreateFile](#) function.

## TTID

Data type: `uint32`

Access type: Read-only

Qualifiers: `WmiDataId(2)`, `Pointer`

Thread identifier of the thread that is creating the file.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[FileIo](#)

# FileIo\_DirEnum class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for the enumerate directory and directory notification events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{72, 77}, EventTypeName{"DirEnum", "DirNotify"}]
class FileIo_DirEnum : FileIo
{
    uint32 IrpPtr;
    uint32 TTID;
    uint32 FileObject;
    uint32 FileKey;
    uint32 Length;
    uint32 InfoClass;
    uint32 FileIndex;
    string FileName;
};
```

## Members

The **FileIo\_DirEnum** class has these types of members:

- [Properties](#)

## Properties

The **FileIo\_DirEnum** class has these properties.

### FileIndex

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(7)**

File index from which to continue directory enumeration.

## **FileKey**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Pointer

To determine the directory name, match the value of this property to the **FileObject** property of a [FileIo\\_Name](#) event.

## **FileName**

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(8), StringTermination("NullTerminated"), Format("w")

Pattern specified for directory enumeration.

## **FileObject**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Identifier that can be used for correlating operations to the same opened file object instance between file create and close events.

## **InfoClass**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6), Pointer

Requested directory enumeration information class.

## **IrpPtr**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

IO request packet. This property identifies the IO activity.

### Length

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5)

Size of the query buffer, in bytes.

### TTID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Thread identifier of the thread that is performing the operation.

## Remarks

Directory enumeration and directory notification events are recorded when a directory is enumerated or a directory change notification is sent to registered listeners, respectively.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[FileIo](#)

# FileIo\_Info class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for file information event.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{69, 70, 71, 74, 75}, EventTypeName{"SetInfo", "Delete", "Rename",  
"QueryInfo", "FSControl"}]  
class FileIo_Info : FileIo  
{  
    uint32 IrpPtr;  
    uint32 TTID;  
    uint32 FileObject;  
    uint32 FileKey;  
    uint32 ExtraInfo;  
    uint32 InfoClass;  
};
```

## Members

The **FileIo\_Info** class has these types of members:

- [Properties](#)

## Properties

The **FileIo\_Info** class has these properties.

### ExtraInfo

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5), Pointer

For FileDispositionInformation requests, this field contains the requested disposition. For FileEndOfFileInformation and FileAllocationInformation requests, this field contains the specified file size.

## **FileKey**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Pointer

To determine the file name, match the value of this property to the **FileObject** property of a [\*\*FileIo\\_Name\*\*](#) event.

## **FileObject**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Identifier that can be used for correlating operations to the same opened file object instance between file create and close events.

## **InfoClass**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6)

Requested file information class.

## **IrpPtr**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

IO request packet. This property identifies the IO activity.

## **TTID**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Thread identifier of the thread that is performing the operation.

## Remarks

Set information and query information events indicate that the file attributes were set or queried. A file system control (FSControl) event is recorded when a FSCTL command is issued.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[FileIo](#)

# FileIo\_Name class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for file I/O events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{0, 32, 35, 36}, EventTypeName{"Name", "FileCreate", "FileDelete",
"FileRundown"}]
class FileIo_Name : FileIo
{
    uint32 FileObject;
    string FileName;
};
```

## Members

The **FileIo\_Name** class has these types of members:

- [Properties](#)

## Properties

The **FileIo\_Name** class has these properties.

FileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(2), StringTermination("NullTerminated"), Format("w")

Full path to the file, not including the drive letter.

FileObject

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Match the value of this pointer to the **FileObject** pointer value in a [DiskIo\\_TypeGroup1](#) event to determine the type of I/O operation.

## Remarks

**Windows Server 2003:** To retrieve the drive letter for the file name path, use the **FileObject** property value to map to the corresponding [DiskIo\\_TypeGroup1](#) event. From the **DiskIo\_TypeGroup1** event, use the **DiskNumber** and **ByteOffset** property values to map to the corresponding [SystemConfig\\_LogDisk](#) event (**ByteOffset** maps to **StartOffset**). The **DriveLetterString** property contains the drive letter.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[FileIo](#)

# FileIo\_OpEnd class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for file operation end events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{76}, EventTypeName{"OperationEnd"}]
class FileIo_OpEnd : FileIo
{
    uint32 IrpPtr;
    uint32 ExtraInfo;
    uint32 NtStatus;
};
```

## Members

The **FileIo\_OpEnd** class has these types of members:

- [Properties](#)

## Properties

The **FileIo\_OpEnd** class has these properties.

### ExtraInfo

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Extra information returned by the file system for the operation. For example for a read request, the actual number of bytes that were read.

### IrpPtr

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

IO request packet. This property identifies the IO activity that is ending.

### NtStatus

Data type: `uint32`

Access type: Read-only

Qualifiers: WmiDataId(3)

Return value from the operation.

## Remarks

[FileIo](#) events are logged at the beginning of the operation. OpEnd events can be enabled separately to indicate the end of those operations. Irp can be used to correlate the begin and end events.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[FileIo](#)

# FileIo\_ReadWrite class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for file read and write events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{67, 68}, EventTypeName{"Read", "Write"}]
class FileIo_ReadWrite : FileIo
{
    uint64 Offset;
    uint32 IrpPtr;
    uint32 TTID;
    uint32 FileObject;
    uint32 FileKey;
    uint32 IoSize;
    uint32 IoFlags;
};
```

## Members

The **FileIo\_ReadWrite** class has these types of members:

- [Properties](#)

## Properties

The **FileIo\_ReadWrite** class has these properties.

### FileKey

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5), Pointer

To determine the file name, match the value of this property to the **FileObject** property of a [FileIo\\_Name](#) event.

## **FileObject**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Pointer

Identifier that can be used for correlating operations to the same opened file object instance between file create and close events.

## **IoFlags**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7)

IO request packet flags specified for this operation.

## **IoSize**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6)

Number of bytes requested.

## **IrpPtr**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

IO request packet. This property identifies the IO activity.

## **Offset**

Data type: **uint64**

Access type: Read-only

Qualifiers: WmiDataId(1)

Starting file offset for the requested operation.

### TTID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Thread identifier of the thread that is performing the operation.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[FileIo](#)

# FileIo\_SimpleOp class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for simple file operation events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{65, 66, 73}, EventTypeName{"Cleanup", "Close", "Flush"}]
class FileIo_SimpleOp : FileIo
{
    uint32 IrpPtr;
    uint32 TTID;
    uint32 FileObject;
    uint32 FileKey;
};
```

## Members

The **FileIo\_SimpleOp** class has these types of members:

- [Properties](#)

## Properties

The **FileIo\_SimpleOp** class has these properties.

### FileKey

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Pointer

To determine the file name, match the value of this property to the **FileObject** property of a [FileIo\\_Name](#) event.

### FileObject

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Identifier that can be used for correlating operations to the same opened file object instance between file create and close events.

### IrpPtr

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

IO request packet. This property identifies the IO activity.

### TTID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Thread identifier of the thread that is performing the operation.

## Remarks

The Cleanup event is logged when the last handle to the file is closed. The Close event specifies that a file object is being freed. The Flush event specifies when the file buffers are fully flushed to disk.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[FileIo](#)



# FileIo\_V0 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for file I/O events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{90cbdc39-4a3e-11d1-84f4-0000f80464e3"}), EventVersion(0)]
class FileIo_V0 : MSNT_SystemTrace
{
};
```

## Members

The **FileIo\_V0** class does not define any members.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[MSNT\\_SystemTrace](#)

[FileIo](#)

[FileIo\\_V0\\_Name](#)

[FileIo\\_V1](#)

# FileIo\_V0\_Name class

Article • 01/07/2021 • 2 minutes to read

The **FileIo\_V0\_Name** class is an older version of the event type class for file I/O events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(0), EventTypeName("Name")]
class FileIo_V0_Name : FileIo_V0
{
    uint32 FileObject;
    string FileName;
};
```

## Members

The **FileIo\_V0\_Name** class has these types of members:

- [Properties](#)

## Properties

The **FileIo\_V0\_Name** class has these properties.

FileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(2), StringTermination("NullTerminated"), Format("w")

Full path to the file, not including the drive letter.

FileObject

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Match the value of this pointer to the **FileObject** pointer value in a **DiskIo\_TypeGroup1** event to determine the type of I/O operation.

## Remarks

**Windows Server 2003:** To retrieve the drive letter for the file name path, use the **FileObject** property value to map to the corresponding **DiskIo\_TypeGroup1** event. From the **DiskIo\_TypeGroup1** event, use the **DiskNumber** and **ByteOffset** property values to map to the corresponding **SystemConfig\_LogDisk** event. The **DriveLetterString** property contains the drive letter.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[FileIo\\_V0](#)

# FileIo\_V1 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for file I/O events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{90cbdc39-4a3e-11d1-84f4-0000f80464e3"}), EventVersion(1)]
class FileIo_V1 : MSNT_SystemTrace
{
};
```

## Members

The **FileIo\_V1** class does not define any members.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[FileIo](#)

[FileIo\\_V0](#)

[FileIo\\_V1\\_Name](#)

# FileIo\_V1\_Name class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for file I/O events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{0, 32}, EventTypeName{"Name", "FileCreate"}]
class FileIo_V1_Name : FileIo
{
    uint32 FileObject;
    string FileName;
};
```

## Members

The **FileIo\_V1\_Name** class has these types of members:

- [Properties](#)

## Properties

The **FileIo\_V1\_Name** class has these properties.

FileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(2), StringTermination("NullTerminated"), Format("w")

Full path to the file, not including the drive letter.

FileObject

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Match the value of this pointer to the **FileObject** pointer value in a [Disklo\\_TypeGroup1](#) event to determine the type of I/O operation.

## Remarks

**Windows Server 2003:** To retrieve the drive letter for the file name path, use the **FileObject** property value to map to the corresponding [Disklo\\_TypeGroup1](#) event. From the **Disklo\_TypeGroup1** event, use the **DiskNumber** and **ByteOffset** property values to map to the corresponding [SystemConfig\\_LogDisk](#) event (**ByteOffset** maps to **StartOffset**). The **DriveLetterString** property contains the drive letter.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[FileIo](#)

# HWConfig class

Article • 01/07/2021 • 2 minutes to read

The **HWConfig** class is the parent class for hardware configuration events on Windows XP.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{01853a65-418f-4f36-aefc-dc0f1d2fd235}")]
class HWConfig : MSNT_SystemTrace
{
};
```

## Members

The **HWConfig** class does not define any members.

## Remarks

These events provide the hardware configuration of the computer. Unlike other NT Kernel Logger events, the kernel session automatically generates hardware configuration events; you do not enable these events when starting the NT Kernel Logger session.

**Windows 2000:** Not supported.

For hardware configuration events on Windows Vista and Windows Server 2003, see the [SystemConfig](#) class.

Event trace consumers can implement special processing for hardware configuration events by calling the [SetTraceCallback](#) function and specifying [EventTraceConfigGuid](#) as the *pGuid* parameter. Use the following event types to identify the actual hardware configuration event when consuming events.

| Event type | Description |
|------------|-------------|
|------------|-------------|

| Event type   | Description   |
|--|---|
| EVENT_TRACE_TYPE_CONFIG_CPU(Event type value is 10)          | CPU configuration event. The <a href="#">HWConfig_CPU</a> MOF classes defines the event data for this event.                        |
| EVENT_TRACE_TYPE_CONFIG_LOGICALDISK(Event type value is 12)  | Logical disk configuration event. The <a href="#">HWConfig_LogDisk</a> MOF classes MOF class defines the event data for this event. |
| EVENT_TRACE_TYPE_CONFIG_NIC(Event type value is 13)          | NIC configuration event. The <a href="#">HWConfig_NIC</a> MOF classes defines the event data for this event.                        |
| EVENT_TRACE_TYPE_CONFIG_PHYSICALDISK(Event type value is 11) | Physical disk configuration event. The <a href="#">HWConfig_PhysDisk</a> MOF classes defines the event data for this event.         |

## Requirements

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |
| Minimum supported server | None supported                 |

## See also

[MSNT\\_SystemTrace](#)

[HWConfig\\_CPU](#)

[HWConfig\\_LogDisk](#)

[HWConfig\\_NIC](#)

[HWConfig\\_PhysDisk](#)

# HWConfig\_CPU class

Article • 01/07/2021 • 2 minutes to read

The **HWConfig\_CPU** class is the event type class for CPU configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(10), EventTypeName("CPU")]
class HWConfig_CPU : HWConfig
{
    uint32 MHz;
    uint32 NumberOfProcessors;
    uint32 MemSize;
    uint32 PageSize;
    uint32 AllocationGranularity;
    string ComputerName;
};
```

## Members

The **HWConfig\_CPU** class has these types of members:

- [Properties](#)

## Properties

The **HWConfig\_CPU** class has these properties.

AllocationGranularity

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(5)**

Granularity with which virtual memory is allocated.

ComputerName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(6), StringTermination("NullTerminated"), Format("w")

Name of the computer.

MemSize

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Total amount of physical memory available to the operating system.

MHz

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Maximum speed of the processor, in megahertz.

NumberOfProcessors

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Number of processors on the computer.

PageSize

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

Size of a swap page, in bytes.

# Requirements

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |
| Minimum supported server | None supported                 |

## See also

[HWConfig](#)

# HWConfig\_LogDisk class

Article • 01/07/2021 • 2 minutes to read

The **HWConfig\_LogDisk** class is the event type class for logical disk configuration events.

The following syntax is simplified from MOF code.

## Syntax

```
syntax

[EventType(12), EventTypeName("LogDisk")]
class HWConfig_LogDisk : HWConfig
{
    uint32 DiskNumber;
    uint32 Pad;
    uint64 StartOffset;
    uint64 PartitionSize;
};
```

## Members

The **HWConfig\_LogDisk** class has these types of members:

- [Properties](#)

## Properties

The **HWConfig\_LogDisk** class has these properties.

DiskNumber

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Index number of the disk containing this partition.

Pad

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Reserved.

PartitionSize

Data type: **uint64**

Access type: Read-only

Qualifiers: WmiDataId(4)

Total size of the partition, in bytes.

StartOffset

Data type: **uint64**

Access type: Read-only

Qualifiers: WmiDataId(3)

Starting offset (in bytes) of the partition from the beginning of the disk.

## Requirements

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |
| Minimum supported server | None supported                 |

## See also

[HWConfig](#)

# HWConfig\_NIC class

Article • 01/07/2021 • 2 minutes to read

The **HWConfig\_NIC** class is the event type class for network interface card configuration events.

The following syntax is simplified from MOF code.

## Syntax

```
syntax

[EventType(13), EventTypeName("NIC")]
class HWConfig_NIC : HWConfig
{
    string NICName;
};
```

## Members

The **HWConfig\_NIC** class has these types of members:

- [Properties](#)

## Properties

The **HWConfig\_NIC** class has these properties.

NICName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(1), StringTermination("NullTerminated"), Format("w")

Name of the network interface card.

## Requirements

| Requirement | Value |
|-------------|-------|
|-------------|-------|

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |
| Minimum supported server | None supported                 |

## See also

[HWConfig](#)

# HWConfig\_PhysDisk class

Article • 01/07/2021 • 2 minutes to read

The **HWConfig\_PhysDisk** class is the event type class for physical disk configuration events.

The following syntax is simplified from MOF code.

## Syntax

```
syntax

[EventType(11), EventTypeName("PhysDisk")]
class HWConfig_PhysDisk : HWConfig
{
    uint32 DiskNumber;
    uint32 BytesPerSector;
    uint32 SectorsPerTrack;
    uint32 TracksPerCylinder;
    uint64 Cylinders;
    uint32 SCSIPort;
    uint32 SCSCIPort;
    uint32 SCSIPath;
    uint32 SCSCIPath;
    uint32 SCSITarget;
    uint32 SCSCITarget;
    uint32 SCSIUn;
    string Manufacturer;
};
```

## Members

The **HWConfig\_PhysDisk** class has these types of members:

- [Properties](#)

## Properties

The **HWConfig\_PhysDisk** class has these properties.

BytesPerSector

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Number of bytes in each sector for the physical disk drive.

Cylinders

Data type: **uint64**

Access type: Read-only

Qualifiers: WmiDataId(5)

Total number of cylinders on the physical disk drive. Note: the value for this property is obtained through extended functions of BIOS interrupt 13h. The value may be inaccurate if the drive uses a translation scheme to support high capacity disk sizes. Consult the manufacturer for accurate drive specifications.

DiskNumber

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Index number of the disk containing this partition.

Manufacturer

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(10), StringTermination("NullTerminated"), Format("w")

Name of the disk drive manufacturer.

SCSILun

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(9)

SCSI logical unit number (LUN) of the SCSI adapter.

SCSIPath

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7)

SCSI bus number of the SCSI adapter.

SCSIPort

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6)

SCSI number of the SCSI adapter.

SCSITarget

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(8)

Contains the number of the target device.

SectorsPerTrack

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Number of sectors in each track for this physical disk drive.

TracksPerCylinder

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

Number of tracks in each cylinder on the physical disk drive. Note: the value for this property is obtained through extended functions of BIOS interrupt 13h. The value may be inaccurate if the drive uses a translation scheme to support high capacity disk sizes. Consult the manufacturer for accurate drive specifications.

# Requirements

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |
| Minimum supported server | None supported                 |

## See also

[HWConfig](#)

# Image class

Article • 01/07/2021 • 2 minutes to read

This class is the parent class for image events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{2cb15d1d-5fc1-11d2-abe1-00a0c911f518}"), EventVersion(2)]
class Image : MSNT_SystemTrace
{
};
```

## Members

The **Image** class does not define any members.

## Remarks

To enable image events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_IMAGE\_LOAD** flag in the **EnableFlags** member of the **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function.

Event trace consumers can implement special processing for image load events by calling the [SetTraceCallback](#) function and specifying **ImageLoadGuid** as the *pGuid* parameter. Use the following event types to identify image load events when consuming events.

| Event type  | Description  |
|---|--|
| <b>EVENT_TRACE_TYPE_LOAD</b> (Event type value is 10) | Image load event. Generated when a DLL or executable file is loaded. The provider generates only one event for the first time a given DLL is loaded. The <a href="#">Image_Load</a> MOF class defines the event data for this event. |

| Event type                                       | Description   |
|--|---|
| EVENT_TRACE_TYPE_END(Event type value is 2)      | Image unload event. Generated when a DLL or executable file is unloaded. The provider generates only one event for the last time a given DLL is unloaded. The <a href="#">Image_Load</a> MOF class defines the event data for this event. |
| EVENT_TRACE_TYPE_DC_START(Event type value is 3) | Data collection start event. Enumerates all loaded images at the beginning of the trace. The <a href="#">Image_Load</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_DC_END(Event type value is 4)   | Data collection end event. Enumerates all loaded images at the end of the trace. The <a href="#">Image_Load</a> MOF class defines the event data for this event.  |

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Image\\_Load](#)

[Image\\_V0](#)

[Image\\_V1](#)

# Image\_Load class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for image events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(10, 2, 3, 4), EventTypeName("Load", "Unload", "DCStart",
"DCEnd")]
class Image_Load : Image
{
    uint32 ImageBase;
    uint32 ImageSize;
    uint32 ProcessId;
    uint32 ImageCheckSum;
    uint32 TimeDateStamp;
    uint32 Reserved0;
    uint32 DefaultBase;
    uint32 Reserved1;
    uint32 Reserved2;
    uint32 Reserved3;
    uint32 Reserved4;
    string FileName;
};
```

## Members

The **Image\_Load** class has these types of members:

- [Properties](#)

## Properties

The **Image\_Load** class has these properties.

DefaultBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7), Pointer

Default base address.

FileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(12), StringTermination("NullTerminated"), Format("w")

File name and extension of the DLL or executable image.

ImageBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Base address of the application in which the image is loaded.

ImageCheckSum

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

Image check sum.

ImageSize

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Size of the image being loaded.

When consuming this property, the data type for this property is actually `size_t`. The Pointer qualifier is used to determine if the `size_t` is 4-bytes or 8-bytes.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Identifies the process into which the image is loaded.

Reserved0

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6)

Reserved.

Reserved1

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(8)

Reserved.

Reserved2

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(9)

Reserved.

Reserved3

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(10)

Reserved.

Reserved4

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(11)

Reserved.

TimeDateStamp

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5)

Time and date that the image was loaded or unloaded.

## Remarks

The DCStart and DCEnd events enumerate all loaded images at the beginning and end of the trace, respectively.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Image](#)

[Image\\_V0](#)

[Image\\_V1](#)

# Image\_V0 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for image load events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{2cb15d1d-5fc1-11d2-abe1-00a0c911f518"}), EventVersion(0)]
class Image_V0 : MSNT_SystemTrace
{
};
```

## Members

The **Image\_V0** class does not define any members.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[MSNT\\_SystemTrace](#)

[Image](#)

[Image\\_V0\\_Load](#)

[Image\\_V1](#)

# Image\_V0\_Load class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for image load events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(10), EventTypeName("Load")]
class Image_V0_Load
{
    uint32 BaseAddress;
    uint32 ModuleSize;
    string ImageFileName;
};
```

## Members

The **Image\_V0\_Load** class has these types of members:

- [Properties](#)

## Properties

The **Image\_V0\_Load** class has these properties.

BaseAddress

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(1)**, **Pointer**

Base address of the application in which the image is loaded.

ImageFileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(3), StringTermination("NullTerminated"), Format("w")

File name and extension of the DLL or executable image to load.

ModuleSize

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the image.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[Image\\_V0](#)

# Image\_V1 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for image load events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{2cb15d1d-5fc1-11d2-abe1-00a0c911f518"}), EventVersion(1)]
class Image_V1 : MSNT_SystemTrace
{
};
```

## Members

The **Image\_V1** class does not define any members.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Image](#)

[Image\\_V0](#)

[Image\\_V1\\_Load](#)

# Image\_V1\_Load class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for image load events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(10), EventTypeName("Load")]
class Image_V1_Load : Image_V1
{
    uint32 ImageBase;
    uint32 ImageSize;
    uint32 ProcessId;
    string FileName;
};
```

## Members

The **Image\_V1\_Load** class has these types of members:

- [Properties](#)

## Properties

The **Image\_V1\_Load** class has these properties.

FileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(4), StringTermination("NullTerminated"), Format("w")

File name and extension of the DLL or executable image to load.

ImageBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Base address of the application in which the image is loaded.

### ImageSize

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Size of the image being loaded.

When consuming this property, the data type for this property is actually size\_t. The Pointer qualifier is used to determine if the size\_t is 4-bytes or 8-bytes.

### ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Identifies the process into which the image is loaded.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[Image\\_V1](#)

# Lost\_Event class

Article • 01/07/2021 • 2 minutes to read

This is the parent class for events lost in a real-time session.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{6a399ae0-4bc6-4de9-870b-3657f8947e7e"})]
class Lost_Event : MSNT_SystemTrace
{
};
```

## Members

The **Lost\_Event** class does not define any members.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[RT\\_LostEvent](#)

# RT\_LostEvent class

Article • 01/07/2021 • 2 minutes to read

This event type class is used to indicate that events were lost in a real-time session.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{32, 33, 34}, EventTypeName{"RTLostEvent", "RTLostBuffer",
"RTLostFile"}]
class RT_LostEvent : Lost_Event
{
};
```

## Members

The **RT\_LostEvent** class does not define any members.

## Remarks

The RTLostEvent event type indicates that one or more events were lost. The RTLostBuffer event type indicates that one or more buffers were lost. The RTLostEvent and RTLostBuffer event types are delivered before processing events from the buffer.

The RTLostFile indicates that the backing file used by the AutoLogger to capture events was lost.

Loosing events depends on the frequency with which the events are logged and how much time the consumer spends processing the events.

## Requirements

| Requirement              | Value                             |
|--------------------------|-----------------------------------|
| Minimum supported client | Windows Vista [desktop apps only] |
| Minimum supported server | None supported                    |

## See also

[Lost\\_Event](#)

# ObTrace class

Article • 01/07/2021 • 2 minutes to read

Represents the parent class from which all object manager event trace classes are derived.

The following syntax is simplified from Managed Object Format (MOF) code and includes all of the inherited properties.

## Syntax

### syntax

```
[DisplayName("Object"), Dynamic, EventVersion(2), Guid("{89497f50-effe-4440-8cf2-ce6b1cdcac7}")]
class ObTrace : MSNT_SystemTrace
{  
};
```

## Members

The **ObTrace** class does not define any members.

## Remarks

To enable object manager event tracing, call the [TraceSetInformation](#) function with the *InformationClass* parameter equal to the **TRACE\_INFO\_CLASS** enumeration value **TraceSystemTraceEnableFlagsInfo** and the **EVENT\_TRACE\_PROPERTIES** structure's *EnableFlags* member equal to **PERF\_OB\_HANDLE** (0x80000040).

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows 8 [desktop apps only]           |
| Minimum supported server | Windows Server 2012 [desktop apps only] |
| MOF                      | Wmicore.mof                             |



# ObHandleDuplicateEvent class

Article • 01/07/2021 • 2 minutes to read

Represents the event type class for handle duplication events.

The following syntax is simplified from Managed Object Format (MOF) code and includes all of the inherited properties.

## Syntax

### syntax

```
[Dynamic, EventType(34), EventTypeName("DuplicateHandle")]
class ObHandleDuplicateEvent : ObTrace
{
    uint32 Object;
    uint16 ObjectType;
    uint32 SourceHandle;
    uint32 TargetHandle;
    uint32 TargetProcessId;
};
```

## Members

The **ObHandleDuplicateEvent** class has these types of members:

- [Properties](#)

## Properties

The **ObHandleDuplicateEvent** class has these properties.

### Object

Data type: **uint32**

Access type: Read-only

Qualifiers: [Format \("x"\)](#), [Pointer](#), [WmiDataId](#) (1)

The object.

### ObjectType

Data type: **uint16**

Access type: Read-only

Qualifiers: [WmiDataId](#) (5)

The object type.

### **SourceHandle**

Data type: **uint32**

Access type: Read-only

Qualifiers: [Format](#) ("x"), [WmiDataId](#) (2)

The handle of the source object.

### **TargetHandle**

Data type: **uint32**

Access type: Read-only

Qualifiers: [Format](#) ("x"), [WmiDataId](#) (3)

The handle of the target object.

### **TargetProcessId**

Data type: **uint32**

Access type: Read-only

Qualifiers: [Format](#) ("x"), [WmiDataId](#) (4)

The process identifier of the target.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows 8 [desktop apps only]           |
| Minimum supported server | Windows Server 2012 [desktop apps only] |
| MOF                      | Wmicore.mof                             |



# ObHandleEvent class

Article • 01/07/2021 • 2 minutes to read

Represents the event type class for handle creation or closure events.

The following syntax is simplified from Managed Object Format (MOF) code and includes all of the inherited properties.

## Syntax

### syntax

```
[Dynamic, EventType{32,33}, EventTypeName{CreateHandle,CloseHandle}]
class ObHandleEvent : ObTrace
{
    uint32 Handle;
    uint32 Object;
    string ObjectName;
    uint16 ObjectType;
};
```

## Members

The **ObHandleEvent** class has these types of members:

- [Properties](#)

## Properties

The **ObHandleEvent** class has these properties.

### Handle

Data type: **uint32**

Access type: Read-only

Qualifiers: **Format** ("x"), **WmiDataId** (2)

The object handle.

### Object

Data type: **uint32**

Access type: Read-only

Qualifiers: **Format** ("x"), **Pointer**, **WmiDataId** (1)

The object.

### ObjectName

Data type: **string**

Access type: Read-only

Qualifiers: **Format** ("w"), **StringTermination** ("NullTerminated"), **WmiDataId** (4)

The object name.

### ObjectType

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

The object type.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows 8 [desktop apps only]           |
| Minimum supported server | Windows Server 2012 [desktop apps only] |
| MOF                      | Wmicore.mof                             |

# ObHandleRundownEvent class

Article • 01/07/2021 • 2 minutes to read

Represents the event type class for object handle events related to the beginning and end of data collection. This event involves the enumerating of all handles when rundown is performed.

The following syntax is simplified from Managed Object Format (MOF) code and includes all of the inherited properties.

## Syntax

### syntax

```
[Dynamic, EventType{38,39}, EventTypeName{HandleDCStart,HandleDCEnd}]
class ObHandleRundownEvent : ObTrace
{
    uint32 Handle;
    uint32 Object;
    string ObjectName;
    uint16 ObjectType;
    uint32 ProcessId;
};
```

## Members

The **ObHandleRundownEvent** class has these types of members:

- [Properties](#)

## Properties

The **ObHandleRundownEvent** class has these properties.

### Handle

Data type: **uint32**

Access type: Read-only

Qualifiers: [Format \("x"\)](#), [WmiDataId](#) (3)

The object handle.

## Object

Data type: **uint32**

Access type: Read-only

Qualifiers: **Format** ("x"), **Pointer**, **WmiDataId** (1)

The object.

## ObjectName

Data type: **string**

Access type: Read-only

Qualifiers: **Format** ("w"), **StringTermination** ("NullTerminated"), **WmiDataId** (5)

The object name.

## ObjectType

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

The object type.

## ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: **Format** ("x"), **WmiDataId** (2)

The process identifier.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows 8 [desktop apps only]           |
| Minimum supported server | Windows Server 2012 [desktop apps only] |

| <b>Requirement</b> | <b>Value</b> |
|--------------------|--------------|
| MOF                | Wmicore.mof  |

# ObTypeEvent class

Article • 01/07/2021 • 2 minutes to read

Represents the event type class for object type events related to the beginning and end of data collection. This event involves the mapping of the type index values to the type names.

The following syntax is simplified from Managed Object Format (MOF) code and includes all of the inherited properties.

## Syntax

### syntax

```
[Dynamic, EventType{36,37}, EventTypeName{TypeDCStart,TypeDCEnd}]
class ObTypeEvent : ObTrace
{
    uint16 ObjectType;
    uint16 Reserved;
    string TypeName;
};
```

## Members

The **ObTypeEvent** class has these types of members:

- [Properties](#)

## Properties

The **ObTypeEvent** class has these properties.

### ObjectType

Data type: **uint16**

Access type: Read-only

Qualifiers: [WmiDataId](#) (1)

The object type.

### Reserved

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Reserved.

### TypeName

Data type: **string**

Access type: Read-only

Qualifiers: **Format** ("w"), **StringTermination** ("NullTerminated"), **WmiDataId** (3)

The type name.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows 8 [desktop apps only]           |
| Minimum supported server | Windows Server 2012 [desktop apps only] |
| MOF                      | Wmicore.mof                             |

# PageFault\_V2 class

Article • 01/07/2021 • 2 minutes to read

This class is the parent class for page fault events.

The following syntax is simplified from MOF code.

## Syntax

syntax

```
[Guid("{3d6fa8d3-fe05-11d0-9dda-00c04fd7ba7c}"), EventVersion(2)]
class PageFault_V2 : MSNT_SystemTrace
{
};
```

## Members

The **PageFault\_V2** class does not define any members.

## Remarks

To enable all page fault events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_MEMORY\_PAGEFAULTS** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function. You can also specify the following flags:

- **EVENT\_TRACE\_FLAG\_MEMORY\_HARDFAULTS**
- **EVENT\_TRACE\_FLAG\_VIRTUALALLOC**

Event trace consumers can implement special processing for all page fault events by calling the [SetTraceCallback](#) function and specifying **PageFaultGuid** as the *pGuid* parameter. Use the following event types to identify the actual memory event when consuming events.

| Event type                                      | Description   |
|---|---|
| EVENT_TRACE_TYPE_MM_COW(Event type value is 12) | Copy-on-write event. The <a href="#">PageFault_TypeGroup1</a> MOF class defines the event data for this event. Prior to Windows Vista, the <a href="#">PageFault_TransitionFault</a> MOF class defines the event. |

| Event type                                      | Description   |
|---|---|
| EVENT_TRACE_TYPE_MM_DZF(Event type value is 11) | Demand zero fault event. The <a href="#">PageFault_TypeGroup1</a> MOF class defines the event data for this event. Prior to Windows Vista, the <a href="#">PageFault_TransitionFault</a> MOF class defines the event. |
| EVENT_TRACE_TYPE_MM_GPF(Event type value is 13) | Guard page fault event. The <a href="#">PageFault_TypeGroup1</a> MOF class defines the event data for this event. Prior to Windows Vista, the <a href="#">PageFault_TransitionFault</a> MOF class defines the event.  |
| EVENT_TRACE_TYPE_MM_HPF(Event type value is 14) | Hard page fault event. The <a href="#">PageFault_TypeGroup1</a> MOF class defines the event data for this event. Prior to Windows Vista, the <a href="#">PageFault_TransitionFault</a> MOF class defines the event.   |
| EVENT_TRACE_TYPE_MM_TF(Event type value is 10)  | Transition fault event. The <a href="#">PageFault_TypeGroup1</a> MOF class defines the event data for this event. Prior to Windows Vista, the <a href="#">PageFault_TransitionFault</a> MOF class defines the event.  |
| EVENT_TRACE_TYPE_MM_AV(Event type value is 15)  | Access violation event. The <a href="#">PageFault_TypeGroup1</a> MOF class defines the event data for this event.   |
| Event type value, 32                            | Hard page fault event. The <a href="#">PageFault_HardFault</a> MOF class defines the event data for this event.   |
| Event type value, 105                           | Image load in page file event. The <a href="#">PageFault_ImageLoadBacked</a> MOF class defines the event data for this event.   |
| Event type value, 98                            | Virtual allocation event. The <a href="#">VirtualAlloc</a> MOF class defines the event data for this event.   |
| Event type value, 99                            | Virtual free event. The <a href="#">VirtualAlloc</a> MOF class defines the event data for this event.   |

You can use the **ProcessId** and **ThreadId** members of [EVENT\\_TRACE\\_HEADER](#) to identify the faulting process or thread.

## Requirements

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported server | Windows Server 2003 [desktop apps only] |

# PageFault\_HardFault class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for hard page fault events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{32}, EventTypeName{"HardFault"}]
class PageFault_HardFault : PageFault_V2
{
    object InitialTime;
    uint64 ReadOffset;
    uint32 VirtualAddress;
    uint32 FileObject;
    uint32 TThreadId;
    uint32 ByteCount;
};
```

## Members

The **PageFault\_HardFault** class has these types of members:

- [Properties](#)

## Properties

The **PageFault\_HardFault** class has these properties.

### ByteCount

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(6)**

Amount of data read from ReadOffset to satisfy the fault.

### FileObject

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Pointer

Match the value of this pointer to the **FileObject** pointer value in a **FileIo\_Name** event to determine the name of the file.

### **InitialTime**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(1), Extension("WmiTime")

Start time stamp at which page fault occurred.

### **ReadOffset**

Data type: **uint64**

Access type: Read-only

Qualifiers: WmiDataId(2), Format("x")

File offset from which data was read to satisfy fault.

### **TThreadId**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5), Format("x")

Thread identifier of the thread that encountered the page fault.

### **VirtualAddress**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Faulting address.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[PageFault\\_V2](#)

# PageFault\_ImageLoadBacked class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for image load in page file events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{105}, EventTypeName{"ImageLoadBacked"}]
class PageFault_ImageLoadBacked : PageFault_V2
{
    uint32 FileObject;
    uint32 DeviceChar;
    uint16 FileChar;
    uint16 LoadFlags;
};
```

## Members

The **PageFault\_ImageLoadBacked** class has these types of members:

- [Properties](#)

## Properties

The **PageFault\_ImageLoadBacked** class has these properties.

### DeviceChar

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Format("x")

Reserved.

### FileChar

Data type: **uint16**

Access type: Read-only

Qualifiers: WmiDataId(3), Format("x")

Reserved.

### FileObject

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Match the value of this pointer to the **FileObject** pointer value in a **FileIo\_Name** event to determine the name of the file.

### LoadFlags

Data type: **uint16**

Access type: Read-only

Qualifiers: WmiDataId(4), Format("x")

Reserved.

## Remarks

The event is logged during an image section creation (for example, a DLL load) when the memory manager determines that the image needs to be copied into and run from the page file. Typically, images are run from the source file but pagefile-backing can occur when the memory manager determines that the source file may be modified asynchronously by existing writers or when the source file is on a removable media or a remote device.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[PageFault\\_V2](#)

# PageFault\_TransitionFault class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for page fault events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11, 12, 13, 14}, EventTypeName{"TransitionFault",  
"DemandZeroFault", "CopyOnWrite", "GuardPageFault", "HardPageFault"}]  
class PageFault_TransitionFault : PageFault_V2  
{  
    uint32 VirtualAddress;  
    uint32 ProgramCounter;  
};
```

## Members

The **PageFault\_TransitionFault** class has these types of members:

- [Properties](#)

## Properties

The **PageFault\_TransitionFault** class has these properties.

ProgramCounter

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Pointer to the current instruction being executed.

VirtualAddress

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Virtual address of the page that caused the page fault.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[PageFault\\_V2](#)

# PageFault\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for page fault events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11, 12, 13, 14, 15}, EventTypeName{"TransitionFault",  
"DemandZeroFault", "CopyOnWrite", "GuardPageFault", "HardPageFault",  
"AccessViolation"}]  
class PageFault_TypeGroup1 : PageFault_V2  
{  
    uint32 VirtualAddress;  
    uint32 ProgramCounter;  
};
```

## Members

The **PageFault\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **PageFault\_TypeGroup1** class has these properties.

ProgramCounter

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Pointer to the current instruction being executed.

VirtualAddress

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Virtual address of the page that caused the page fault.

## Remarks

A page fault occurs when a page sought in the memory cache is not found there and must be retrieved from elsewhere in memory (a soft fault) or from disk (a hard fault).

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[PageFault\\_V2](#)

# PageFault\_VirtualAlloc class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for virtual allocation events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{98, 99}, EventTypeName{"VirtualAlloc", "VirtualFree"}]
class PageFault_VirtualAlloc : PageFault_V2
{
    uint32 BaseAddress;
    object RegionSize;
    uint32 ProcessId;
    uint32 Flags;
};
```

## Members

The **PageFault\_VirtualAlloc** class has these types of members:

- [Properties](#)

## Properties

The **PageFault\_VirtualAlloc** class has these properties.

### BaseAddress

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

The starting address at which memory was allocated or freed.

### Flags

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Format("x")

The type of memory allocation that was performed. For possible values, see the *fAllocationType* parameter of the [VirtualAllocEx](#) function.

### ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

The process that owned the memory (can be different from the thread that performed the allocation).

### RegionSize

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(2), Extension("SizeT")

The size, in bytes, of the memory that was allocated or freed.

## Requirements

| Requirement              | Value                                      |
|--------------------------|--|
| Minimum supported client | Windows 7 [desktop apps only]              |
| Minimum supported server | Windows Server 2008 R2 [desktop apps only] |

## See also

[PageFault\\_V2](#)

# PerfInfo class

Article • 01/07/2021 • 2 minutes to read

This class is the parent class for performance counter events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{ce1dbfb4-137e-4da6-87b0-3f59aa102cbc}"), EventVersion(2)]
class PerfInfo : MSNT_SystemTrace
{
};
```

## Members

The **PerfInfo** class does not define any members.

## Remarks

To enable deferred procedure call (DPC) events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_DPC** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function. You can also specify one or more of the following flags:

- **EVENT\_TRACE\_FLAG\_INTERRUPT**
- **EVENT\_TRACE\_FLAG\_PROFILE**
- **EVENT\_TRACE\_FLAG\_SYSTEMCALL**

Event trace consumers can implement special processing for DPC events by calling the [SetTraceCallback](#) function and specifying **PerfInfoGuid** as the *pGuid* parameter. Use the following event types to identify the actual event when consuming events.

| Event type           | Description  |
|----------------------|--|
| Event type value, 46 | Sampled profile event. The <a href="#">SampledProfile</a> MOF class defines the event data for this event. |

| Event type           | Description   |
|----------------------|---|
| Event type value, 51 | System call enter event. The <a href="#">SysCallEnter</a> MOF class defines the event data for this event.      |
| Event type value, 52 | System call exit event. The <a href="#">SysCallExit</a> MOF class defines the event data for this event.        |
| Event type value, 66 | Threaded DPC event. The <a href="#">DPC</a> MOF class defines the event data for this event.                    |
| Event type value, 67 | Interrupt service routine (ISR) event. The <a href="#">ISR</a> MOF class defines the event data for this event. |
| Event type value, 68 | DPC event. The <a href="#">DPC</a> MOF class defines the event data for this event.                             |
| Event type value, 69 | DPC timer event. The <a href="#">DPC</a> MOF class defines the event data for this event.                       |

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# DPC class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for device deferred procedure call (DPC) events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{66, 68, 69}, EventTypeName{"ThreadDPC", "DPC", "TimerDPC"}]
class DPC : PerfInfo
{
    object InitialTime;
    uint32 Routine;
};
```

## Members

The **DPC** class has these types of members:

- [Properties](#)

## Properties

The **DPC** class has these properties.

### InitialTime

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(1), Extension("WmiTime")

DPC entry time.

### Routine

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Address of DPC routine. Use the address with the Image events to find which image started.

## Remarks

These events are logged when a DPC is entered. You use these events to monitor and verify the behavior of drivers and kernel-mode components. For example, you can use DPC, ISR, and Image events to determine those components that spend too much time at high interrupt levels. DPC and ISR events have an entry time stamp which is used to compute the duration of the routines. The image events are read to construct the memory regions that map to certain modules. You can use the mapping to locate the module that contains the interrupt routine.

The TimerDPC event records when a DPC fires as a result of a timer expiration and the ThreadDPC event records when a threaded DPC executes.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# ISR class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for interrupt service routine (ISR) events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{67}, EventTypeName{"ISR"}]
class ISR : PerfInfo
{
    object InitialTime;
    uint32 Routine;
    uint8 ReturnValue;
    uint8 Vector;
    uint16 Reserved;
};
```

## Members

The **ISR** class has these types of members:

- [Properties](#)

## Properties

The **ISR** class has these properties.

### InitialTime

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(1), Extension("WmiTime")

ISR entry time.

### Reserved

Data type: **uint16**

Access type: Read-only

Qualifiers: WmiDataId(5), Pointer

Reserved.

#### **ReturnValue**

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(3)

Boolean indicating if the interrupt was claimed (is True if the interrupt was claimed).

#### **Routine**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Address of ISR routine. Use the address with the Image events to find which image started.

#### **Vector**

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(4)

Vector from interrupt descriptor table to which ISR routine is assigned.

## **Requirements**

| <b>Requirement</b>       | <b>Value</b>                            |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# SampledProfile class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for sampled profile events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{46}, EventTypeName{"SampleProfile"}]
class SampledProfile : PerfInfo
{
    uint32 InstructionPointer;
    uint32 ThreadId;
    uint32 Count;
};
```

## Members

The **SampledProfile** class has these types of members:

- [Properties](#)

## Properties

The **SampledProfile** class has these properties.

### Count

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Not used.

### InstructionPointer

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Address of the image that was running at the time the processor was sampled.

#### ThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Thread identifier of the thread that was running at the time the processor was sampled.

## Remarks

These events provide a sampled execution profile. The event records what was being executed on the processor. You can use the Image events to identify the binary module containing that instruction. You can then use this information to produce an execution profile for the duration of the trace.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# SysCallEnter class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for system call enter events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{51}, EventTypeName{"SysC1Enter"}]
class SysCallEnter : PerfInfo
{
    uint32 SysCallAddress;
};
```

## Members

The **SysCallEnter** class has these types of members:

- [Properties](#)

## Properties

The **SysCallEnter** class has these properties.

### SysCallAddress

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Address of the NT function call that is being entered.

## Requirements

| Requirement              | Value                             |
|--------------------------|-----------------------------------|
| Minimum supported client | Windows Vista [desktop apps only] |

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# SysCallExit class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for system call exit events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{52}, EventTypeName{"SysClExit"}]
class SysCallExit : PerfInfo
{
    uint32 SysCallNtStatus;
};
```

## Members

The **SysCallExit** class has these types of members:

- [Properties](#)

## Properties

The **SysCallExit** class has these properties.

### SysCallNtStatus

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Format("x")

Status code returned by the NT system call.

## Requirements

| Requirement              | Value                             |
|--------------------------|-----------------------------------|
| Minimum supported client | Windows Vista [desktop apps only] |

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# Process class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for process events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{3d6fa8d0-fe05-11d0-9dda-00c04fd7ba7c}"), EventVersion(3)]
class Process : MSNT_SystemTrace
{
};
```

## Members

The **Process** class does not define any members.

## Remarks

To enable process events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_PROCESS** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function. You can also specify the following flag:

- **EVENT\_TRACE\_FLAG\_PROCESS\_COUNTERS**

Event trace consumers can implement special processing for process events by calling the [SetTraceCallback](#) function and specifying **ProcessGuid** as the *pGuid* parameter. Use the following event types to identify the actual process event when consuming events.

| Event type  | Description  |
|---|--|
| <b>EVENT_TRACE_TYPE_END</b> (Event type value is 2)   | End process event. The <a href="#">Process_TypeGroup1</a> MOF class defines the event data for this event.   |
| <b>EVENT_TRACE_TYPE_START</b> (Event type value is 1) | Start process event. The <a href="#">Process_TypeGroup1</a> MOF class defines the event data for this event. |

| Event type          | Description   |
|---------------------|---|
| Event type value, 3 | Start data collection process event. Enumerates processes that are currently running at the time the kernel session starts. The <a href="#">Process_TypeGroup1</a> MOF class defines the event data for this event. |
| Event type value, 4 | End data collection process event. Enumerates processes that are currently running at the time the kernel session ends. The <a href="#">Process_TypeGroup1</a> MOF class defines the event data for this event.     |

Process and thread start events may be logged in the context of the parent process or thread. As a result, the **ProcessId** and **ThreadId** members of [EVENT\\_TRACE\\_HEADER](#) may not correspond to the process and thread being created. This is why these events contain the process and thread identifiers in the event data (in addition to those in the event header).

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps   UWP apps]       |
| Minimum supported server | Windows Server 2008 [desktop apps   UWP apps] |

## See also

[MSNT\\_SystemTrace](#)

[Process\\_TypeGroup1](#)

[Process\\_V0](#)

[Process\\_V1](#)

[Process\\_V2](#)

# Process\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for process events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{1, 2, 3, 4, 39}, EventTypeName{"Start", "End", "DCStart",
"DCEnd", "Defunct"}]
class Process_TypeGroup1 : Process
{
    uint32 UniqueProcessKey;
    uint32 ProcessId;
    uint32 ParentId;
    uint32 SessionId;
    sint32 ExitStatus;
    uint32 DirectoryTableBase;
    object UserID;
    string ImageFileName;
    string CommandLine;
};
```

## Members

The **Process\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Process\_TypeGroup1** class has these properties.

CommandLine

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(9), StringTermination("NullTerminated"), Format("w")

Full command line of the process.

DirectoryTableBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6), Pointer

The physical address of the page table of the process.

ExitStatus

Data type: **sint32**

Access type: Read-only

Qualifiers: WmiDataId(5)

Exit status of the stopped process.

ImageFileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(8), StringTermination("NullTerminated")

Path to the executable file of the process.

ParentId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Format("x")

Unique identifier of the process that creates this process. Process identifier numbers are reused, so they only identify a process for the lifetime of that process. It is possible that the process identified by ParentProcessId is terminated, so ParentProcessId may not refer to a running process. It is also possible that ParentProcessId incorrectly refers to a process that reuses a process identifier.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Format("x")

Global process identifier that you can use to identify a process. The value is valid from the time a process is created until it is terminated.

SessionId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

Unique identifier that an operating system generates when it creates a new session. A session spans a period of time from log on until log off from a specific system.

UniqueProcessKey

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

The address of the process object in the kernel.

UserSID

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(7), Extension("Sid")

Security identifier (SID) for the user context under which the event happens.

## Remarks

The DCStart and DCEnd event types enumerate the processes that are currently running, including idle and system processes, at the time the kernel session starts and ends, respectively.

## Requirements

---

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[Process](#)

# Process\_V0 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for process events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{3d6fa8d0-fe05-11d0-9dda-00c04fd7ba7c}"), EventVersion(0)]
class Process_V0 : MSNT_SystemTrace
{
};
```

## Members

The **Process\_V0** class does not define any members.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[MSNT\\_SystemTrace](#)

[Process](#)

[Process\\_V0\\_TypeGroup1](#)

[Process\\_V1](#)

# Process\_V0\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for process events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{1, 2, 3, 4}, EventTypeName{"Start", "End", "DCStart", "DCEnd"}]
class Process_V0_TypeGroup1 : Process_V0
{
    uint32 ProcessId;
    uint32 ParentId;
    object UserSID;
    string ImageFileName;
};
```

## Members

The **Process\_V0\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Process\_V0\_TypeGroup1** class has these properties.

ImageFileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(4), StringTermination("NullTerminated")

Path to the executable file of the process.

ParentId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Unique identifier of the process that creates a process. Process identifier numbers are reused, so they only identify a process for the lifetime of that process. It is possible that the process identified by ParentProcessId is terminated, so ParentProcessId may not refer to a running process. It is also possible that ParentProcessId incorrectly refers to a process that reuses a process identifier.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Global process identifier that you can use to identify a process. The value is valid from the time a process is created until it is terminated.

UserSID

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("Sid")

Security identifier (SID) for the user context under which the event happens.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[Process\\_V0](#)

# Process\_V1 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for process events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{3d6fa8d0-fe05-11d0-9dda-00c04fd7ba7c}"), EventVersion(1)]
class Process_V1 : MSNT_SystemTrace
{
};
```

## Members

The **Process\_V1** class does not define any members.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Process](#)

[Process\\_V0](#)

[Process\\_V1\\_TypeGroup1](#)

# Process\_V1\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for process events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{1, 2, 3, 4}, EventTypeName{"Start", "End", "DCStart", "DCEnd"}]
class Process_V1_TypeGroup1 : Process_V1
{
    uint32 PageDirectoryBase;
    uint32 ProcessId;
    uint32 ParentId;
    uint32 SessionId;
    sint32 ExitStatus;
    object UserID;
    string ImageFileName;
};
```

## Members

The **Process\_V1\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Process\_V1\_TypeGroup1** class has these properties.

ExitStatus

Data type: **sint32**

Access type: Read-only

Qualifiers: WmiDataId(5)

Exit status of the stopped process.

ImageFileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(7), StringTermination("NullTerminated")

Path to the executable file of the process.

PageDirectoryBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Reserved.

ParentId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Unique identifier of the process that creates this process. Process identifier numbers are reused, so they only identify a process for the lifetime of that process. It is possible that the process identified by ParentProcessId is terminated, so ParentProcessId may not refer to a running process. It is also possible that ParentProcessId incorrectly refers to a process that reuses a process identifier.

**Windows Server 2003:** Includes the Format("x") qualifier.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Global process identifier that you can use to identify a process. The value is valid from the time a process is created until it is terminated.

**Windows Server 2003:** Includes the Format("x") qualifier.

SessionId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

Unique identifier that an operating system generates when it creates a new session. A session spans a period of time from log on until log off from a specific system.

UserSID

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Sid")

Security identifier (SID) for the user context under which the event happens.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[Process\\_V1](#)

[Process\\_V1](#)

# Process\_V2 class

Article • 11/19/2021 • 2 minutes to read

This class is the parent class for process events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{3d6fa8d0-fe05-11d0-9dda-00c04fd7ba7c}"), EventVersion(2)]
class Process_V2 : MSNT_SystemTrace
{
};
```

## Members

The **Process** class does not define any members.

## Remarks

To enable process events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_PROCESS** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function. You can also specify the following flag:

- **EVENT\_TRACE\_FLAG\_PROCESS\_COUNTERS**

Event trace consumers can implement special processing for process events by calling the [SetTraceCallback](#) function and specifying **ProcessGuid** as the *pGuid* parameter. Use the following event types to identify the actual process event when consuming events.

| Event type  | Description   |
|---|---|
| <b>EVENT_TRACE_TYPE_END</b> (Event type value is 2)   | End process event. The <a href="#">Process_V2_TypeGroup1</a> MOF class defines the event data for this event.   |
| <b>EVENT_TRACE_TYPE_START</b> (Event type value is 1) | Start process event. The <a href="#">Process_V2_TypeGroup1</a> MOF class defines the event data for this event. |

| Event type           | Description  |
|----------------------|--|
| Event type value, 3  | Start data collection process event. Enumerates processes that are currently running at the time the kernel session starts. The <a href="#">Process_V2_TypeGroup1</a> MOF class defines the event data for this event. |
| Event type value, 4  | End data collection process event. Enumerates processes that are currently running at the time the kernel session ends. The <a href="#">Process_V2_TypeGroup1</a> MOF class defines the event data for this event.     |
| Event type value, 32 | Performance counters event. The <a href="#">Process_V2_TypeGroup2</a> MOF class defines the event data for this event.   |
| Event type value, 33 | Rundown of the performance counters at the start of the session. The <a href="#">Process_V2_TypeGroup2</a> MOF class defines the event data for this event.  |
| Event type value, 39 | Defunct process event. The <a href="#">Process_V2_TypeGroup1</a> MOF class defines the event data for this event.  |

Process and thread start events may be logged in the context of the parent process or thread. As a result, the **ProcessId** and **ThreadId** members of [EVENT\\_TRACE\\_HEADER](#) may not correspond to the process and thread being created. This is why these events contain the process and thread identifiers in the event data (in addition to those in the event header).

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps   UWP apps]       |
| Minimum supported server | Windows Server 2008 [desktop apps   UWP apps] |

## See also

[MSNT\\_SystemTrace](#)

[Process](#)

[Process\\_TypeGroup1](#)

**Process\_V0**

**Process\_V1**

# Process\_V2\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for process events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{1, 2, 3, 4, 39}, EventTypeName{"Start", "End", "DCStart",
"DCEnd", "Defunct"}]
class Process_V2_TypeGroup1 : Process_V2
{
    uint32 UniqueProcessKey;
    uint32 ProcessId;
    uint32 ParentId;
    uint32 SessionId;
    sint32 ExitStatus;
    object UserSID;
    string ImageFileName;
    string CommandLine;
};
```

## Members

The **Process\_V2\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Process\_V2\_TypeGroup1** class has these properties.

CommandLine

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(8), StringTermination("NullTerminated"), Format("w")

Full command line of the process.

ExitStatus

Data type: **sint32**

Access type: Read-only

Qualifiers: WmiDataId(5)

Exit status of the stopped process.

ImageFileName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(7), StringTermination("NullTerminated")

Path to the executable file of the process.

ParentId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Format("x")

Unique identifier of the process that creates this process. Process identifier numbers are reused, so they only identify a process for the lifetime of that process. It is possible that the process identified by ParentProcessId is terminated, so ParentProcessId may not refer to a running process. It is also possible that ParentProcessId incorrectly refers to a process that reuses a process identifier.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Format("x")

Global process identifier that you can use to identify a process. The value is valid from the time a process is created until it is terminated.

SessionId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

Unique identifier that an operating system generates when it creates a new session. A session spans a period of time from log on until log off from a specific system.

UniqueProcessKey

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

The address of the process object in the kernel.

UserSID

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Sid")

Security identifier (SID) for the user context under which the event happens.

## Remarks

The DCStart and DCEnd event types enumerate the process that are currently running, including idle and system process, at the time the kernel session starts and ends, respectively.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[Process\\_V2](#)



# Process\_V2\_TypeGroup2 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for process counter events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{32, 33}, EventTypeName{"PerfCtr", PerfCtrRundown"}]
class Process_V2_TypeGroup2 : Process_V2
{
    uint32 ProcessId;
    uint32 PageFaultCount;
    uint32 HandleCount;
    uint32 Reserved;
    object PeakVirtualSize;
    object PeakWorkingSetSize;
    object PeakPagefileUsage;
    object QuotaPeakPagedPoolUsage;
    object QuotaPeakNonPagedPoolUsage;
    object VirtualSize;
    object WorkingSetSize;
    object PagefileUsage;
    object QuotaPagedPoolUsage;
    object QuotaNonPagedPoolUsage;
    object PrivatePageCount;
};
```

## Members

The **Process\_V2\_TypeGroup2** class has these types of members:

- [Properties](#)

## Properties

The **Process\_V2\_TypeGroup2** class has these properties.

### HandleCount

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Count of used handles.

### **PageFaultCount**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Count of page faults.

### **PagefileUsage**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(12), Extension("SizeT")

Current page file usage.

### **PeakPagefileUsage**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(7), Extension("SizeT")

Largest page file size used.

### **PeakVirtualSize**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("SizeT")

Largest virtual page size used.

### **PeakWorkingSetSize**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("SizeT")

Largest working set size used.

### **PrivatePageCount**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(15), Extension("SizeT")

Current private physical page count.

### **ProcessId**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Format("x")

Global process identifier that you can use to identify a process. The value is valid from the time a process is created until it is terminated.

### **QuotaNonPagedPoolUsage**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(14), Extension("SizeT")

Current committed non-paged memory usage.

### **QuotaPagedPoolUsage**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(13), Extension("SizeT")

Current committed paged memory usage.

### **QuotaPeakNonPagedPoolUsage**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(9), Extension("SizeT")

Largest committed non-paged memory used.

### **QuotaPeakPagedPoolUsage**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(8), Extension("SizeT")

Largest committed paged memory used.

### **Reserved**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

Reserved.

### **VirtualSize**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(10), Extension("SizeT")

Current virtual page size.

### **WorkingSetSize**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(11), Extension("SizeT")

Current working set size.

## Remarks

These events are logged when the process ends. The event indicates how a process handled memory usage.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Process\\_V2](#)

# Registry class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for registry events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{ae53722e-c863-11d2-8659-00c04fa321a1"}), EventVersion(2)]
class Registry : MSNT_SystemTrace
{
};
```

## Members

The **Registry** class does not define any members.

## Remarks

To enable registry events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_REGISTRY** in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function.

Event trace consumers can implement special processing for registry events by calling the [SetTraceCallback](#) function and specifying **RegistryGuid** as the *pGuid* parameter. Use the following event types to identify the actual registry event when consuming events.

| Event type   | Description  |
|--|--|
| <b>EVENT_TRACE_TYPE_REGCREATE</b> (Event type value is 10) | Create key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event. |
| <b>EVENT_TRACE_TYPE_REGDELETE</b> (Event type value is 12) | Delete key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event. |

| Event type   | Description   |
|--|---|
| EVENT_TRACE_TYPE_REGDELETEVALUE(Event type value is 15)        | Delete value event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_REGENUMERATEKEY(Event type value is 17)       | Enumerate key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_REGENUMERATEVALUEKEY(Event type value is 18)  | Enumerate value key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_REGFLUSH(Event type value is 21)              | Flush key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_REGKCBDM(P(Event type value is 22)            | Create key event. Generated when a registry operation uses handles rather than strings to reference subkeys. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event. |
| EVENT_TRACE_TYPE_REGOPEN(Event type value is 11)               | Open key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_REGQUERY(Event type value is 13)              | Query key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_REGQUERYMULTIPLEVALUE(Event type value is 19) | Query multiple value event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_REGQUERYVALUE(Event type value is 16)         | Query value event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_REGSETINFORMATION(Event type value is 20)     | Set information event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_REGSETVALUE(Event type value is 14)           | Set value event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |

| Event type           | Description   |
|----------------------|---|
| Event type value, 23 | Delete key event. Generated when a registry operation uses handles rather than strings to reference subkeys. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event. |
| Event type value, 24 | Enumerates the registry keys open at the beginning of the session. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| Event type value, 25 | Enumerates the registry keys open at the end of the session. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| Event type value, 26 | The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |
| Event type value, 27 | Open key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Registry\\_TypeGroup1](#)

[Registry\\_V0](#)

[Registry\\_V1](#)



# Registry\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for registry events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27}, EventTypeName{"Create", "Open", "Delete", "Query", "SetValue",
"DeleteValue", "QueryValue", "EnumerateKey", "EnumerateValueKey",
"QueryMultipleValue", "SetInformation", "Flush", "KCBCreate", "KCBDelete",
"KCBRundownBegin", "KCBRundownEnd", "Virtualize", "Close"}]
class Registry_TypeGroup1 : Registry
{
    sint64 InitialTime;
    uint32 Status;
    uint32 Index;
    uint32 KeyHandle;
    string KeyName;
};
```

## Members

The **Registry\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Registry\_TypeGroup1** class has these properties.

Index

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(3)**

The subkey index for the registry operation (such as **EnumerateKey**).

InitialTime

Data type: **sint64**

Access type: Read-only

Qualifiers: WmiDataId(1)

Initial time of the registry operation.

KeyHandle

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Pointer

Handle to the registry key.

KeyName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(5), StringTermination("NullTerminated"), Format("w")

Name of the registry key.

Status

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

NTSTATUS value of the registry operation.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Registry](#)

# Registry\_V0 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for registry events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{ae53722e-c863-11d2-8659-00c04fa321a1"}), EventVersion(0)]
class Registry_V0 : MSNT_SystemTrace
{
};
```

## Members

The **Registry\_V0** class does not define any members.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Registry](#)

[Registry\\_V0\\_TypeGroup1](#)

[Registry\\_V1](#)

# Registry\_V0\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for registry events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21},  
EventTypeName{"Create", "Open", "Delete", "Query", "SetValue",  
"DeleteValue", "QueryValue", "EnumerateKey", "EnumerateValueKey",  
"QueryMultipleValue", "SetInformation", "Flush"}]  
class Registry_V0_TypeGroup1 : Registry_V0  
{  
    uint32 Status;  
    uint32 KeyHandle;  
    sint64 ElapsedTime;  
    string KeyName;  
};
```

## Members

The **Registry\_V0\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Registry\_V0\_TypeGroup1** class has these properties.

ElapsedTime

Data type: **sint64**

Access type: Read-only

Qualifiers: WmiDataId(3)

Elapsed time of the registry operation.

KeyHandle

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Handle to the registry key.

**KeyName**

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(4), StringTermination("NullTerminated"), Format("w")

Name of the registry key.

**Status**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

NTSTATUS value of the registry operation.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[Registry\\_V0](#)

# Registry\_V1 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for registry events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{ae53722e-c863-11d2-8659-00c04fa321a1"}), EventVersion(1)]
class Registry_V1 : MSNT_SystemTrace
{
};
```

## Members

The **Registry\_V1** class does not define any members.

## Remarks

To enable registry events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_REGISTRY** in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function.

Event trace consumers can implement special processing for registry events by calling the [SetTraceCallback](#) function and specifying **RegistryGuid** as the *pGuid* parameter. Use the following event types to identify the actual registry event when consuming events.

| Event type   | Description  |
|--|--|
| <b>EVENT_TRACE_TYPE_REGCREATE</b> (Event type value is 10) | Create key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event. |
| <b>EVENT_TRACE_TYPE_REGDELETE</b> (Event type value is 12) | Delete key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event. |

| Event type   | Description  |
|--|--|
| EVENT_TRACE_TYPE_REGDELETEVALUE(Event type value is 15)        | Delete value event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_REGENUMERATEKEY(Event type value is 17)       | Enumerate key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_REGENUMERATEVALUEKEY(Event type value is 18)  | Enumerate value key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_REGFLUSH(Event type value is 21)              | Flush key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_REGKCBDM(P(Event type value is 22)            | Generated when a registry operation uses handles rather than strings to reference subkeys. These events are recorded once for each handle—the first time the handle is referenced. Repeated references to the handle do not generate multiple events.<br>The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.<br><b>Windows 2000:</b> This value is not supported. |
| EVENT_TRACE_TYPE_REGOPEN(Event type value is 11)               | Open key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_REGQUERY(Event type value is 13)              | Query key event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_REGQUERYMULTIPLEVALUE(Event type value is 19) | Query multiple value event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_REGQUERYVALUE(Event type value is 16)         | Query value event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_REGSETINFORMATION(Event type value is 20)     | Set information event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event.  |

| Event type   | Description   |
|--|---|
| EVENT_TRACE_TYPE_REGSetValue(Event type value is 14) | Set value event. The <a href="#">Registry_TypeGroup1</a> MOF class defines the event data for this event. |

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Registry](#)

[Registry\\_V0](#)

[Registry\\_V1\\_TypeGroup1](#)

# Registry\_V1\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for registry events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22},  
EventTypeName{"Create", "Open", "Delete", "Query", "SetValue",  
"DeleteValue", "QueryValue", "EnumerateKey", "EnumerateValueKey",  
"QueryMultipleValue", "SetInformation", "Flush", "RunDown"}]  
class Registry_V1_TypeGroup1 : Registry  
{  
    uint32 Status;  
    uint32 KeyHandle;  
    sint64 ElapsedTime;  
    uint32 Index;  
    string KeyName;  
};
```

## Members

The **Registry\_V1\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Registry\_V1\_TypeGroup1** class has these properties.

ElapsedTime

Data type: **sint64**

Access type: Read-only

Qualifiers: WmiDataId(3)

Elapsed time of the registry operation.

## Index

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

The subkey index for the registry operation (such as EnumerateKey).

## KeyHandle

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Handle to the registry key.

## KeyName

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(5), StringTermination("NullTerminated"), Format("w")

Name of the registry key.

## Status

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

NTSTATUS value of the registry operation.

# Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[Registry](#)

[Registry\\_V1](#)

# SplitIo class

Article • 01/07/2021 • 2 minutes to read

This class is the parent class for split IO events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{d837ca92-12b9-44a5-ad6a-3a65b3578aa8}")]
class SplitIo : MSNT_SystemTrace
{
};
```

## Members

The **SplitIo** class does not define any members.

## Remarks

To enable split IO events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_SPLIT\_IO** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function.

Event trace consumers can implement special processing for split IO events by calling the [SetTraceCallback](#) function and specifying **SplitIoGuid** as the *pGuid* parameter. Use the following event type to identify the actual event when consuming events.

| Event type              | Description   |
|-------------------------|---|
| Event type value,<br>32 | Split IO event. The <a href="#">SplitIo_Info</a> MOF class defines the event data for this event. |

Split IO events indicate that the IO requests have been split into multiple disk IO requests due to the underlying mirroring disk hardware.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# SplitIo\_Info class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for split IO events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{32}, EventTypeName{"VolMgr"}]
class SplitIo_Info : SplitIo
{
    uint32 ParentIrp;
    uint32 ChildIrp;
};
```

## Members

The **SplitIo\_Info** class has these types of members:

- [Properties](#)

## Properties

The **SplitIo\_Info** class has these properties.

### ChildIrp

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Pointer

Child IO request packet.

### ParentIrp

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Pointer

Parent IO request packet.

## Remarks

Indicates that the volume manager split the IRP into two IRPs.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

# StackWalk class

Article • 01/07/2021 • 2 minutes to read

This class is the parent class for stack tracing events.

The following syntax is simplified from MOF code and includes all inherited properties.

## Syntax

### syntax

```
[Guid("{def2fe46-7bd6-4b80-bd94-f57fe20d0ce3}")]
class StackWalk : MSNT_SystemTrace
{
};
```

## Members

The **StackWalk** class does not define any members.

## Remarks

To enable stack tracing of kernel events, call the [TraceSetInformation](#) function and specify the kernel events and types for which you want to capture the stack trace. To enable stack tracing for other events, set the **EnableProperty** member of [ENABLE\\_TRACE\\_PARAMETERS](#) to [EVENT\\_ENABLE\\_PROPERTY\\_STACK\\_TRACE](#).

Use the following event type to identify the actual event when consuming events.

| Event type           | Description   |
|----------------------|---|
| Event type value, 32 | Stack tracing event. The <a href="#">StackWalk_Event</a> MOF class defines the event data for this event. |

## Requirements

| Requirement | Value |
|-------------|-------|
|-------------|-------|

| Requirement              | Value                                      |
|--------------------------|--|
| Minimum supported client | Windows 7 [desktop apps only]              |
| Minimum supported server | Windows Server 2008 R2 [desktop apps only] |

# StackWalk\_Event class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for stack tracing events.

The following syntax is simplified from MOF code and includes all inherited properties.

## Syntax

### syntax

```
[EventType{32}, EventTypeName{"Stack"}]
class StackWalk_Event : StackWalk
{
    uint64 EventTimeStamp;
    uint32 StackProcess;
    uint32 StackThread;
    uint32 Stack1;
    uint32 Stack192;
};
```

## Members

The **StackWalk\_Event** class has these types of members:

- [Properties](#)

## Properties

The **StackWalk\_Event** class has these properties.

### EventTimeStamp

Data type: **uint64**

Access type: Read-only

Qualifiers: WmiDataId(1)

Original event time stamp from the event header. Use this time stamp to match the stack to an event.

### Stack1

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), pointer

Address of the call.

### **Stack192**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(195), pointer

Address of the call.

### **StackProcess**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), format("x")

The process identifier of the original event.

### **StackThread**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

The thread identifier of the original event.

## **Remarks**

Note that the class does not show all of the **Stackn** properties that exist between **Stack1** and **Stack192**. Use the size of the event to determine how many **Stackn** properties contain valid addresses.

## **Requirements**

---

| Requirement              | Value                                      |
|--------------------------|--|
| Minimum supported client | Windows 7 [desktop apps only]              |
| Minimum supported server | Windows Server 2008 R2 [desktop apps only] |

# SystemConfig class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for hardware configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{01853a65-418f-4f36-aefc-dc0f1d2fd235}"), EventVersion(2)]
class SystemConfig : MSNT_SystemTrace
{
};
```

## Members

The **SystemConfig** class does not define any members.

## Remarks

These events provide the hardware configuration of the computer. Unlike other NT Kernel Logger events, the kernel session automatically generates hardware configuration events; you do not enable these events when starting the NT Kernel Logger session.

For hardware configuration events on Windows XP, see the [HWConfig](#) class.

Event trace consumers can implement special processing for hardware configuration events by calling the [SetTraceCallback](#) function and specifying [EventTraceConfigGuid](#) as the *pGuid* parameter. Use the following event types to identify the actual hardware configuration event when consuming events.

| Event type  | Description  |
|---|--|
| EVENT_TRACE_TYPE_CONFIG_CPU(Event type value is 10) | CPU configuration event. The <a href="#">SystemConfig_CPU</a> MOF classes defines the event data for this event. |

| Event type   | Description   |
|--|---|
| EVENT_TRACE_TYPE_CONFIG_IDECHANNEL(Event type value is 23)   | Primary and secondary IDE channel configuration event. The <a href="#">SystemConfig_IDEChannel</a> MOF classes MOF class defines the event data for this event. |
| EVENT_TRACE_TYPE_CONFIG_IRQ(Event type value is 21)          | Interrupt request (IRQ) event. The <a href="#">SystemConfig_IRQ</a> MOF classes MOF class defines the event data for this event.                                |
| EVENT_TRACE_TYPE_CONFIG_LOGICALDISK(Event type value is 12)  | Logical disk configuration event. The <a href="#">SystemConfig_LogDisk</a> MOF classes MOF class defines the event data for this event.                         |
| EVENT_TRACE_TYPE_CONFIG_NETINFO(Event type value is 17)      | Network information event. The <a href="#">SystemConfig_Network</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_CONFIG_NIC(Event type value is 13)          | NIC configuration event. The <a href="#">SystemConfig_NIC</a> MOF classes defines the event data for this event.  |
| EVENT_TRACE_TYPE_CONFIG_PHYSICALDISK(Event type value is 11) | Physical disk configuration event. The <a href="#">SystemConfig_PhysDisk</a> MOF classes defines the event data for this event.                                 |
| EVENT_TRACE_TYPE_CONFIG_PNP(Event type value is 22)          | Plug and play event. The <a href="#">SystemConfig_PNP</a> MOF classes MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_CONFIG_POWER(Event type value is 16)        | Power configuration event. The <a href="#">SystemConfig_Power</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_CONFIG_SERVICES(Event type value is 15)     | Services configuration event. The <a href="#">SystemConfig_Services</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_CONFIG_VIDEO(Event type value is 14)        | Graphics adapter configuration event. The <a href="#">SystemConfig_Video</a> MOF class defines the event data for this event.                                   |

## Requirements

| Requirement | Value |
|-------------|-------|
|             |       |

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[SystemConfig\\_CPU](#)

[SystemConfig\\_IDEChannel](#)

[SystemConfig\\_IRQ](#)

[SystemConfig\\_LogDisk](#)

[SystemConfig\\_Network](#)

[SystemConfig\\_NIC](#)

[SystemConfig\\_PhysDisk](#)

[SystemConfig\\_PNP](#)

[SystemConfig\\_Power](#)

[SystemConfig\\_Services](#)

[SystemConfig\\_Video](#)

[SystemConfig\\_V0](#)

# SystemConfig\_CPU class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for CPU configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(10), EventTypeName("CPU")]
class SystemConfig_CPU : SystemConfig
{
    uint32 MHz;
    uint32 NumberOfProcessors;
    uint32 MemSize;
    uint32 PageSize;
    uint32 AllocationGranularity;
    char16 ComputerName[];
    char16 DomainName[];
    uint32 HyperThreadingFlag;
};
```

## Members

The **SystemConfig\_CPU** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_CPU** class has these properties.

### AllocationGranularity

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Granularity with which virtual memory is allocated.

## **ComputerName**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (6), **Max** (256), **Format("s")**

Name of the computer.

## **DomainName**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (7), **Max** (132), **Format("s")**

Name of the domain in which the computer is a member.

## **HyperThreadingFlag**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (8), Pointer

Indicates if the hyper-threading option is on or off for a processor. Each bit reflects the hyper-threading state of a CPU on the computer.

## **MemSize**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Total amount of physical memory available to the operating system.

## **MHz**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Maximum speed of the processor, in megahertz.

### **NumberOfProcessors**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Number of processors on the computer.

### **PageSize**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Size of a swap page, in bytes.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_IDEChannel class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for IDE channel events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(23), EventTypeName("IDEChannel")]
class SystemConfig_IDEChannel : SystemConfig
{
    uint32 TargetId;
    uint32 DeviceType;
    uint32 DeviceTimingMode;
    uint32 LocationInformationLen;
    string LocationInformation;
};
```

## Members

The **SystemConfig\_IDEChannel** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_IDEChannel** class has these properties.

**DeviceTimingMode**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Format("x")

The mode in which the IDE could function. The following are the possible values:

- **PIO\_MODE0** (0x1)
- **PIO\_MODE1** (0x2)
- **PIO\_MODE2** (0x4)

- PIO\_MODE3 (0x8)
- PIO\_MODE4 (0x10)
- SWDMA\_MODE0 (0x20)
- SWDMA\_MODE1 (0x40)
- SWDMA\_MODE2 (0x80)
- MWDMA\_MODE0 (0x100)
- MWDMA\_MODE2 (0x200)
- MWDMA\_MODE3 (0x400)
- UDMA\_MODE0 (0x800)
- UDMA\_MODE1 (0x1000)
- UDMA\_MODE2 (0x2000)
- UDMA\_MODE3 (0x4000)
- UDMA\_MODE4 (0x8000)
- UDMA\_MODE5 (0x10000)

## DeviceType

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Format("x")

The device type. The following are the possible values:

- ATA (1)
- ATAPI (2)

## LocationInformation

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(5), StringTermination("NullTerminated"), Format("w")

The IDE channel (for example, Primary Channel, Secondary Channel, and so on).

## LocationInformationLen

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

Length of the **LocationInformation** string.

### TargetId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

The identifier of the disk.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_IRQ class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for interrupt request (IRQ) events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(21), EventTypeName("IRQ")]
class SystemConfig_IRQ : SystemConfig
{
    uint64 IRQAffinity;
    uint32 IRQNum;
    uint32 DeviceDescriptionLen;
    string DeviceDescription;
};
```

## Members

The **SystemConfig\_IRQ** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_IRQ** class has these properties.

### DeviceDescription

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(4), StringTermination("NullTerminated"), Format("w")

Description of the device or software making the request.

### DeviceDescriptionLen

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Length, in characters, of the string in **DeviceDescription**.

### **IRQAffinity**

Data type: **uint64**

Access type: Read-only

Qualifiers: WmiDataId(1), Format("x")

IRQ affinity mask. The affinity mask identifies the specific processors (or groups of processors) that can receive the IRQ.

### **IRQNum**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Interrupt request line number.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_LogDisk class

Article • 08/25/2021 • 2 minutes to read

This class is the event type class for logical disk configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(12), EventTypeName("LogDisk")]
class SystemConfig_LogDisk : SystemConfig
{
    uint64 StartOffset;
    uint64 PartitionSize;
    uint32 DiskNumber;
    uint32 Size;
    uint32 DriveType;
    char16 DriveLetterString[];
    uint32 Pad1;
    uint32 PartitionNumber;
    uint32 SectorsPerCluster;
    uint32 BytesPerSector;
    uint32 Pad2;
    sint64 NumberOfFreeClusters;
    sint64 TotalNumberOfClusters;
    char16 FileSystem;
    uint32 VolumeExt;
    uint32 Pad3;
};
```

## Members

The `SystemConfig_LogDisk` class has these types of members:

- [Properties](#)

## Properties

The `SystemConfig_LogDisk` class has these properties.

### BytesPerSector

Data type: `uint32`

Access type: Read-only

Qualifiers: **WmiDataId** (10)

Number of bytes in each sector for the physical disk drive.

#### DiskNumber

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Index number of the disk containing this partition.

#### DriveLetterString

Data type: **char16 array**

Access type: Read-only

Qualifiers: **WmiDataId** (6), **Max** (4), **Format("s")**

Drive letter of the disk in the form, "<letter>:".

#### DriveType

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Type of disk drive. Possible values are:

| Value | Meaning                              |
|-------|--------------------------------------|
| 1     | Partition                            |
| 2     | Volume                               |
| 3     | Extended partition on multiple disks |

#### FileSystem

Data type: **char16**

Access type: Read-only

Qualifiers: **WmiDataId** (14), **Max** (16), **Format("s")**

File system on the logical disk, for example, NTFS.

#### **NumberOfFreeClusters**

Data type: **sint64**

Access type: Read-only

Qualifiers: **WmiDataId** (12)

Number of free clusters in the specified volume.

#### **Pad1**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (7)

Not used.

#### **Pad2**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (11)

Not used.

#### **Pad3**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (16)

Not used.

#### **PartitionNumber**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (8)

Index number of the partition.

#### **PartitionSize**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Total size of the partition, in bytes.

#### **SectorsPerCluster**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (9)

Number of sectors in the volume.

#### **Size**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Size of the disk drive, in bytes.

#### **StartOffset**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Starting offset (in bytes) of the partition from the beginning of the disk.

#### **TotalNumberOfClusters**

Data type: **sint64**

Access type: Read-only

Qualifiers: **WmiDataId** (13)

Number of used and free clusters in the volume.

### VolumeExt

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (15)

Reserved.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_Network class

Article • 11/29/2021 • 2 minutes to read

This class is the event type class for network events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(17), EventTypeName("Network")]
class SystemConfig_Network : SystemConfig
{
    uint32 TcbTablePartitions;
    uint32 MaxHashTableSize;
    uint32 MaxUserPort;
    uint32 TcpTimedWaitDelay;
};
```

## Members

The `SystemConfig_Network` class has these types of members:

- [Properties](#)

## Properties

The `SystemConfig_Network` class has these properties.

### MaxHashTableSize

Data type: `uint32`

Access type: Read-only

Qualifiers: `WmiDataId(2)`

The size of the hash table in which TCP control blocks (TCBs) are stored. TCP stores control blocks in a hash table so it can find them very quickly.

### MaxUserPort

Data type: `uint32`

Access type: Read-only

Qualifiers: WmiDataId(3)

The highest port number TCP can assign when an application requests an available user port from the system. Typically, ephemeral ports (those used briefly) are allocated to port numbers 1024 through 5000.

The value for the highest user port number TCP can assign is controlled by a registry setting. For more information, see [MaxUserPort](#).

### TcbTablePartitions

Data type: `uint32`

Access type: Read-only

Qualifiers: WmiDataId(1)

The number of partitions in the Transport Control Block table. Partitioning the Transport Control Block table minimizes contention for table access. This is especially useful on multiprocessor systems.

### TcpTimedWaitDelay

Data type: `uint32`

Access type: Read-only

Qualifiers: WmiDataId(4)

The time that must elapse before TCP can release a closed connection and reuse its resources. This interval between closure and release is known as the TIME\_WAIT state or 2MSL state. During this time, the connection can be reopened at much less cost to the client and server than establishing a new connection.

RFC 793 published by the IETF requires that TCP maintains a closed connection for an interval at least equal to twice the maximum segment lifetime (2MSL) of the network. When a connection is released, its socket pair and TCP control block (TCB) can be used to support another connection. By default, the MSL is defined to be 120 seconds, and the value of this entry is equal to two MSLs, or 4 minutes. For more information, see [RFC 793](#).

Reducing the value of this entry using a registry setting allows TCP to release closed connections faster, providing more resources for new connections. However, if the value

is too low, TCP might release connection resources before the connection is complete, requiring the server to use additional resources to reestablish the connection.

Normally, TCP does not release closed connections until the value of this entry expires. However, TCP can release connections before this value expires if it is running out of TCP control blocks (TCBs). The number of TCBs the system creates is controlled by a registry setting. For more information, see [MaxFreeTCBs](#).

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_NIC class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for network interface card configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(13), EventTypeName("NIC")]
class SystemConfig_NIC : SystemConfig
{
    uint64 PhysicalAddr;
    uint32 PhysicalAddrLen;
    uint32 Ipv4Index;
    uint32 Ipv6Index;
    string NICDescription;
    string IpAddresses;
    string DnsServerAddresses;
};
```

## Members

The `SystemConfig_NIC` class has these types of members:

- [Properties](#)

## Properties

The `SystemConfig_NIC` class has these properties.

DnsServerAddresses

Data type: `string`

Access type: Read-only

Qualifiers: `WmiDataId(7)`, `StringTermination("NullTerminated")`, `Format("w")`

IP addresses to be used in querying for DNS servers. The list of addresses is comma-delimited.

**IpAddresses**

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(6), StringTermination("NullTerminated"), Format("w")

IP addresses associated with the network interface card. The list of addresses is comma-delimited.

**Ipv4Index**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Adapter index for IPv4 NIC. The adapter index may change when an adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

**Ipv6Index**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4)

Adapter index for IPv6 NIC. The adapter index may change when an adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

**NICDescription**

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(5), StringTermination("NullTerminated"), Format("w")

Description of the adapter.

**PhysicalAddr**

Data type: **uint64**

Access type: Read-only

Qualifiers: WmiDataId(1), Format("x")

Hardware address for the adapter.

PhysicalAddrLen

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Length of the hardware address for the adapter.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_PhysDisk class

Article • 08/25/2021 • 2 minutes to read

This class is the event type class for physical disk configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(11), EventTypeName("PhysDisk")]
class SystemConfig_PhysDisk : SystemConfig
{
    uint32 DiskNumber;
    uint32 BytesPerSector;
    uint32 SectorsPerTrack;
    uint32 TracksPerCylinder;
    uint64 Cylinders;
    uint32 SCSIPort;
    uint32 SCSSIPath;
    uint32 SCSSITarget;
    uint32 SCSSLun;
    char16 Manufacturer[];
    uint32 PartitionCount;
    uint8 WriteCacheEnabled;
    uint8 Pad;
    char16 BootDriveLetter[];
    char16 Spare[];
};
```

## Members

The `SystemConfig_PhysDisk` class has these types of members:

- [Properties](#)

## Properties

The `SystemConfig_PhysDisk` class has these properties.

### BootDriveLetter

Data type: `char16` array

Access type: Read-only

Qualifiers: **WmiDataId** (14), **Max** (3), **Format("s")**

Drive letter of the boot drive in the form, "<letter>:".

### **BytesPerSector**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Number of bytes in each sector for the physical disk drive.

### **Cylinders**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Total number of cylinders on the physical disk drive. Note: the value for this property is obtained through extended functions of BIOS interrupt 13h. The value may be inaccurate if the drive uses a translation scheme to support high capacity disk sizes. Consult the manufacturer for accurate drive specifications.

### **DiskNumber**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Index number of the disk containing this partition.

### **Manufacturer**

Data type: **char16 array**

Access type: Read-only

Qualifiers: **WmiDataId** (10), **Max** (256), **Format("s")**

Name of the disk drive manufacturer.

## **Pad**

Data type: **uint8**

Access type: Read-only

Qualifiers: **WmiDataId** (13)

Not used.

## **PartitionCount**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (11)

Number of partitions on this physical disk drive that are recognized by the operating system.

## **SCSILun**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (9)

SCSI logical unit number (LUN) of the SCSI adapter.

## **SCSIPath**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (7)

SCSI bus number of the SCSI adapter.

## **SCSIPort**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (6)

SCSI number of the SCSI adapter.

### **SCSITarget**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (8)

Contains the number of the target device.

### **SectorsPerTrack**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Number of sectors in each track for this physical disk drive.

### **Spare**

Data type: **char16 array**

Access type: Read-only

Qualifiers: **WmiDataId** (15), **Max** (2), **Format("s")**

Not used.

### **TracksPerCylinder**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Number of tracks in each cylinder on the physical disk drive. Note: the value for this property is obtained through extended functions of BIOS interrupt 13h. The value may be inaccurate if the drive uses a translation scheme to support high capacity disk sizes. Consult the manufacturer for accurate drive specifications.

### **WriteCacheEnabled**

Data type: **uint8**

Access type: Read-only

Qualifiers: **WmiDataId** (12)

True if the write cache is enabled.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_PnP class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for PnP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(22), EventTypeName("PnP")]
class SystemConfig_PnP : SystemConfig
{
    uint32 IDLength;
    uint32 DescriptionLength;
    uint32 FriendlyNameLength;
    string DeviceID;
    string DeviceDescription;
    string FriendlyName;
};
```

## Members

The **SystemConfig\_PnP** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_PnP** class has these properties.

### DescriptionLength

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(2)**

Length, in characters, of the **DeviceDescription** string.

### DeviceDescription

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(5), StringTermination("NullTerminated"), Format("w")

Description of the PnP device.

### **DeviceID**

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(4), StringTermination("NullTerminated"), Format("w")

Identifies the PnP device.

### **FriendlyName**

Data type: **string**

Access type: Read-only

Qualifiers: WmiDataId(6), StringTermination("NullTerminated"), Format("w")

Name of the PnP device to use in a user interface.

### **FriendlyNameLength**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3)

Length, in characters, of the FriendlyName string.

### **IDLength**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Length, in characters, of the DeviceID string.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_Power class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for power configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(16), EventTypeName("Power")]
class SystemConfig_Power : SystemConfig
{
    uint8 s1;
    uint8 s2;
    uint8 s3;
    uint8 s4;
    uint8 s5;
    uint8 Pad1;
    uint8 Pad2;
    uint8 Pad3;
};
```

## Members

The **SystemConfig\_Power** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_Power** class has these properties.

Pad1

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(6)

Not used.

Pad2

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(7)

Not used.

Pad3

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(8)

Not used.

s1

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(1)

True indicates the system supports sleep state S1.

s2

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(2)

True indicates the system supports sleep state S2.

s3

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(3)

True indicates the system supports sleep state S3.

s4

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(4)

True indicates the system supports sleep state S4.

s5

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(5)

True indicates the system supports sleep state S5.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_Services class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for service configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(15), EventTypeName("Services")]
class SystemConfig_Services : SystemConfig
{
    uint32 ProcessId;
    uint32 ServiceState;
    uint32 SubProcessTag;
    string ServiceName[];
    string DisplayName[];
    string ProcessName[];
};
```

## Members

The **SystemConfig\_Services** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_Services** class has these properties.

### DisplayName

Data type: **string** array

Access type: Read-only

Qualifiers: **WmiDataId** (5), **StringTermination("NullTerminated")**, **Format("w")**

Display name of the service. The name is case-preserved in the Service Control Manager. However, display name comparisons are always case-insensitive.

### ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (1), **Format("x")**

Identifier of the process in which the service runs.

#### **ProcessName**

Data type: **string** array

Access type: Read-only

Qualifiers: **WmiDataId** (6), **StringTermination("NullTerminated")**, **Format("w")**

Name of the process in which the service runs.

#### **ServiceName**

Data type: **string** array

Access type: Read-only

Qualifiers: **WmiDataId** (4), **StringTermination("NullTerminated")**, **Format("w")**

Unique identifier of the service. The identifier provides an indication of the functionality the service provides.

#### **ServiceState**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (2), **Format("x")**

Current state of the service. For possible values, see the **dwCurrentState** member of **SERVICE\_STATUS\_PROCESS**.

#### **SubProcessTag**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3), **Format("x")**

Identifies the service.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_Video class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for video configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(14), EventTypeName("Video")]
class SystemConfig_Video : SystemConfig
{
    uint32 MemorySize;
    uint32 XResolution;
    uint32 YResolution;
    uint32 BitsPerPixel;
    uint32 VRefresh;
    char16 ChipType[];
    char16 DACType[];
    char16 AdapterString[];
    char16 BiosString[];
    char16 DeviceId[];
    uint32 StateFlags;
};
```

## Members

The **SystemConfig\_Video** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_Video** class has these properties.

### AdapterString

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (8), **Max** (256), **Format("s")**

Name or description of the adapter.

### **BiosString**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (9), **Max** (256), **Format("s")**

BIOS name of the adapter.

### **BitsPerPixel**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Number of bits used to display each pixel.

### **ChipType**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (6), **Max** (256), **Format("s")**

Chip name of the adapter.

### **DACType**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (7), **Max** (256), **Format("s")**

Digital-to-analog converter (DAC) chip name of the adapter.

### **DeviceId**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (10), **Max** (256), **Format("s")**

Address or other identifying information to uniquely name the logical device.

## MemorySize

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Maximum amount of memory supported, in bytes.

## StateFlags

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (11), **Format("x")**

Device state flags. It can be any reasonable combination of the following.

| Value   | Meaning   |
|---|---|
| DISPLAY_DEVICE_ATTACHED_TO_DESKTOP<br>1 (0x1)       | The device is part of the desktop.  |
| DISPLAY_DEVICE_MIRRORING_DRIVER<br>8 (0x8)          | Represents a pseudo device used to mirror application drawing for connecting to a remote computer or other purposes. An invisible pseudo monitor is associated with this device. For example, NetMeeting uses it. |
| DISPLAY_DEVICE_MODESPRUNED<br>134217728 (0x8000000) | The device has more display modes than its output devices support.  |
| DISPLAY_DEVICE_PRIMARY_DEVICE<br>4 (0x4)            | The primary desktop is on the device. For a system with a single display card, this is always set. For a system with multiple display cards, only one device can have this set.                                   |
| DISPLAY_DEVICE_REMOVABLE<br>32 (0x20)               | The device is removable; it cannot be the primary display.  |
| DISPLAY_DEVICE_VGA_COMPATIBLE<br>16 (0x10)          | The device is VGA compatible.   |

## VRefresh

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Current refresh rate, in hertz.

### XResolution

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Current number of horizontal pixels.

### YResolution

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Current number of vertical pixels.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[SystemConfig](#)

# SystemConfig\_V0 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for hardware configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{01853a65-418f-4f36-aefc-dc0f1d2fd235}"), EventVersion(0)]
class SystemConfig_V0 : MSNT_SystemTrace
{
};
```

## Members

The **SystemConfig\_V0** class does not define any members.

## Remarks

For hardware configuration events on Windows XP, see the [HWConfig](#) class.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | None supported                          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[SystemConfig](#)

[SystemConfig\\_V0\\_CPU](#)

[\*\*SystemConfig\\_V0\\_LogDisk\*\*](#)

[\*\*SystemConfig\\_V0\\_NIC\*\*](#)

[\*\*SystemConfig\\_V0\\_PhysDisk\*\*](#)

[\*\*SystemConfig\\_V0\\_Power\*\*](#)

[\*\*SystemConfig\\_V0\\_Services\*\*](#)

[\*\*SystemConfig\\_V0\\_Video\*\*](#)

# SystemConfig\_V0\_CPU class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for CPU configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(10), EventTypeName("CPU")]
class SystemConfig_V0_CPU : SystemConfig_V0
{
    uint32 MHz;
    uint32 NumberOfProcessors;
    uint32 MemSize;
    uint32 PageSize;
    uint32 AllocationGranularity;
    char16 ComputerName[];
    char16 DomainName[];
};
```

## Members

The `SystemConfig_V0_CPU` class has these types of members:

- [Properties](#)

## Properties

The `SystemConfig_V0_CPU` class has these properties.

### AllocationGranularity

Data type: `uint32`

Access type: Read-only

Qualifiers: `WmiDataId` (5)

Granularity with which virtual memory is allocated.

### ComputerName

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (6), **Max** (256)

Name of the computer.

#### **DomainName**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (7), **Max** (132)

Name of the domain in which the computer is a member.

#### **MemSize**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Total amount of physical memory available to the operating system.

#### **MHz**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Maximum speed of the processor, in megahertz.

#### **NumberOfProcessors**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Number of processors on the computer.

#### **PageSize**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Size of a swap page, in bytes.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | None supported                          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[SystemConfig\\_V0](#)

# SystemConfig\_V0\_LogDisk class

Article • 08/25/2021 • 2 minutes to read

This class is the event type class for logical disk configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(12), EventTypeName("LogDisk")]
class SystemConfig_V0_LogDisk : SystemConfig_V0
{
    uint64 StartOffset;
    uint64 PartitionSize;
    uint32 DiskNumber;
    uint32 Size;
    uint32 DriveType;
    char16 DriveLetterString[];
    uint32 Pad;
    uint32 PartitionNumber;
    uint32 SectorsPerCluster;
    uint32 BytesPerSector;
    sint64 NumberOfFreeClusters;
    sint64 TotalNumberOfClusters;
    char16 FileSystem;
    uint32 VolumeExt;
};
```

## Members

The **SystemConfig\_V0\_LogDisk** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_V0\_LogDisk** class has these properties.

### BytesPerSector

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (10)

Number of bytes in each sector for the physical disk drive.

### DiskNumber

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Index number of the disk containing this partition.

### DriveLetterString

Data type: **char16 array**

Access type: Read-only

Qualifiers: **WmiDataId** (6), **Max** (4)

Drive letter of the disk in the form, "<letter>:".

### DriveType

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Type of disk drive. Possible values are:

| Value | Meaning                              |
|-------|--------------------------------------|
| 1     | Partition                            |
| 2     | Volume                               |
| 3     | Extended partition on multiple disks |

### FileSystem

Data type: **char16**

Access type: Read-only

Qualifiers: **WmiDataId** (13), **Max** (16)

File system on the logical disk, for example, NTFS.

### **NumberOfFreeClusters**

Data type: **sint64**

Access type: Read-only

Qualifiers: **WmiDataId** (11)

Number of free clusters in the specified volume.

### **Pad**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (7)

Reserved.

### **PartitionNumber**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (8)

Index number of the partition.

### **PartitionSize**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Total size of the partition, in bytes.

### **SectorsPerCluster**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (9)

Number of sectors in the volume.

### Size

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Size of the disk drive, in bytes.

### StartOffset

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Starting offset (in bytes) of the partition from the beginning of the disk.

### TotalNumberOfClusters

Data type: **sint64**

Access type: Read-only

Qualifiers: **WmiDataId** (12)

Number of used and free clusters in the volume.

### VolumeExt

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (14)

Reserved.

## Requirements

| Requirement | Value |
|-------------|-------|
|-------------|-------|

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | None supported                          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[SystemConfig\\_V0](#)

# SystemConfig\_V0\_NIC class

Article • 01/07/2021 • 3 minutes to read

This class is the event type class for network interface card configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(13), EventTypeName("NIC")]
class SystemConfig_V0_NIC : SystemConfig_V0
{
    char16 NICName[];
    uint32 Index;
    uint32 PhysicalAddrLen;
    char16 PhysicalAddr;
    uint32 Size;
    sint32 IpAddress;
    sint32 SubnetMask;
    sint32 DhcpServer;
    sint32 Gateway;
    sint32 PrimaryWinsServer;
    sint32 SecondaryWinsServer;
    sint32 DnsServer1;
    sint32 DnsServer2;
    sint32 DnsServer3;
    sint32 DnsServer4;
    uint32 Data;
};
```

## Members

The `SystemConfig_V0_NIC` class has these types of members:

- [Properties](#)

## Properties

The `SystemConfig_V0_NIC` class has these properties.

### Data

Data type: `uint32`

Access type: Read-only

Qualifiers: **WmiDataId** (16)

Data field.

### DhcpServer

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (8)

IP address of the dynamic host configuration protocol (DHCP) server. A value of 255.255.255.255 indicates the DHCP server could not be reached, or is in the process of being reached. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

### DnsServer1

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (12)

First server IP addresses to be used in querying for DNS servers. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

### DnsServer2

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (13)

Second server IP addresses to be used in querying for DNS servers. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

### DnsServer3

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (14)

Third server IP addresses to be used in querying for DNS servers. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

#### DnsServer4

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (15)

Fourth server IP addresses to be used in querying for DNS servers. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

#### Gateway

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (9)

IP address of default gateway that the computer system uses. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

#### Index

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Adapter index. The adapter index may change when an adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

#### IpAddress

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (6)

IP addresses associated with the network interface card. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

### **NICName**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (1), **Max** (256)

Name of the network interface card.

### **PhysicalAddr**

Data type: **char16**

Access type: Read-only

Qualifiers: **WmiDataId** (4), **Max** (8)

Hardware address for the adapter.

### **PhysicalAddrLen**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Length of the hardware address for the adapter.

### **PrimaryWinsServer**

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (10)

IP address for the primary WINS server. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

### **SecondaryWinsServer**

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (11)

IP address for the secondary WINS server. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

## Size

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Size, in bytes, of the Data property.

## SubnetMask

Data type: **sint32**

Access type: Read-only

Qualifiers: **WmiDataId** (7)

Subnet mask associated with the network interface card. Each byte of the sint32 represents one of the four parts of the IP address (p1.p2.p3.p4). The low-order byte contains the value for p1, the next byte contains the value for p2, and so on.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | None supported                          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[SystemConfig\\_V0](#)



# SystemConfig\_V0\_PhysDisk class

Article • 08/25/2021 • 2 minutes to read

This class is the event type class for physical disk configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(11), EventTypeName("PhyDisk")]
class SystemConfig_V0_PhysDisk : SystemConfig_V0
{
    uint32 DiskNumber;
    uint32 BytesPerSector;
    uint32 SectorsPerTrack;
    uint32 TracksPerCylinder;
    uint64 Cylinders;
    uint32 SCSIPort;
    uint32 SCSSIPath;
    uint32 SCSSITarget;
    uint32 SCSSILun;
    char16 Manufacturer[];
    uint32 PartitionCount;
    boolean WriteCacheEnabled;
    char16 BootDriveLetter[];
};
```

## Members

The `SystemConfig_V0_PhysDisk` class has these types of members:

- [Properties](#)

## Properties

The `SystemConfig_V0_PhysDisk` class has these properties.

### BootDriveLetter

Data type: `char16` array

Access type: Read-only

Qualifiers: **WmiDataId** (13), **Max** (3)

Drive letter of the boot drive in the form, "<letter>:".

### **BytesPerSector**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Number of bytes in each sector for the physical disk drive.

### **Cylinders**

Data type: **uint64**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Total number of cylinders on the physical disk drive. Note: the value for this property is obtained through extended functions of BIOS interrupt 13h. The value may be inaccurate if the drive uses a translation scheme to support high capacity disk sizes. Consult the manufacturer for accurate drive specifications.

### **DiskNumber**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Index number of the disk containing this partition.

### **Manufacturer**

Data type: **char16 array**

Access type: Read-only

Qualifiers: **WmiDataId** (10), **Max** (256)

Name of the disk drive manufacturer.

### **PartitionCount**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (11)

Number of partitions on this physical disk drive that are recognized by the operating system.

### **SCSILun**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (9)

SCSI logical unit number (LUN) of the SCSI adapter.

### **SCSIPath**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (7)

SCSI bus number of the SCSI adapter.

### **SCSIPort**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (6)

SCSI number of the SCSI adapter.

### **SCSITarget**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (8)

Contains the number of the target device.

## **SectorsPerTrack**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Number of sectors in each track for this physical disk drive.

## **TracksPerCylinder**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Number of tracks in each cylinder on the physical disk drive. Note: the value for this property is obtained through extended functions of BIOS interrupt 13h. The value may be inaccurate if the drive uses a translation scheme to support high capacity disk sizes. Consult the manufacturer for accurate drive specifications.

## **WriteCacheEnabled**

Data type: **boolean**

Access type: Read-only

Qualifiers: **WmiDataId** (12)

True if the write cache is enabled.

# **Requirements**

| <b>Requirement</b>       | <b>Value</b>                            |
|--------------------------|---|
| Minimum supported client | None supported                          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## **See also**

[SystemConfig\\_V0](#)



# SystemConfig\_V0\_Power class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for power configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(16), EventTypeName("Power")]
class SystemConfig_V0_Power : SystemConfig_V0
{
    boolean s1;
    boolean s2;
    boolean s3;
    boolean s4;
    boolean s5;
    uint8 Pad1;
    uint8 Pad2;
    uint8 Pad3;
};
```

## Members

The **SystemConfig\_V0\_Power** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_V0\_Power** class has these properties.

Pad1

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(6)

Reserved.

Pad2

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(7)

Reserved.

Pad3

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(8)

Reserved.

s1

Data type: **boolean**

Access type: Read-only

Qualifiers: WmiDataId(1)

True indicates the system supports sleep state S1.

s2

Data type: **boolean**

Access type: Read-only

Qualifiers: WmiDataId(2)

True indicates the system supports sleep state S2.

s3

Data type: **boolean**

Access type: Read-only

Qualifiers: WmiDataId(3)

True indicates the system supports sleep state S3.

s4

Data type: **boolean**

Access type: Read-only

Qualifiers: WmiDataId(4)

True indicates the system supports sleep state S4.

s5

Data type: **boolean**

Access type: Read-only

Qualifiers: WmiDataId(5)

True indicates the system supports sleep state S5.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | None supported                          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[SystemConfig\\_V0](#)

# SystemConfig\_V0\_Services class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for service configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(15), EventTypeName("Services")]
class SystemConfig_V0_Services : SystemConfig_V0
{
    char16 ServiceName[];
    char16 DisplayName[];
    char16 ProcessName[];
    uint32 ProcessId;
};
```

## Members

The **SystemConfig\_V0\_Services** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_V0\_Services** class has these properties.

### DisplayName

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (2), **Max** (256)

Display name of the service. The name is case-preserved in the Service Control Manager. However, display name comparisons are always case-insensitive.

### ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Identifier of the process in which the service runs.

### **ProcessName**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (3), **Max** (34)

Name of the process in which the service runs.

### **ServiceName**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (1), **Max** (34)

Unique identifier of the service. The identifier provides an indication of the functionality the service provides.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | None supported                          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[SystemConfig\\_V0](#)

# SystemConfig\_V0\_Video class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for video configuration events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType(14), EventTypeName("Video")]
class SystemConfig_V0_Video : SystemConfig_V0
{
    uint32 MemorySize;
    uint32 XResolution;
    uint32 YResolution;
    uint32 BitsPerPixel;
    uint32 VRefresh;
    char16 ChipType[];
    char16 DACType[];
    char16 AdapterString[];
    char16 BiosString[];
    char16 DeviceId[];
    uint32 StateFlags;
};
```

## Members

The **SystemConfig\_V0\_Video** class has these types of members:

- [Properties](#)

## Properties

The **SystemConfig\_V0\_Video** class has these properties.

### AdapterString

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (8), **Max** (256)

Name or description of the adapter.

### **BiosString**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (9), **Max** (256)

BIOS name of the adapter.

### **BitsPerPixel**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (4)

Number of bits used to display each pixel.

### **ChipType**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (6), **Max** (256)

Chip name of the adapter.

### **DACType**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (7), **Max** (256)

Digital-to-analog converter (DAC) chip name of the adapter.

### **DeviceId**

Data type: **char16** array

Access type: Read-only

Qualifiers: **WmiDataId** (10), **Max** (256)

Address or other identifying information to uniquely name the logical device.

## MemorySize

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (1)

Maximum amount of memory supported, in bytes.

## StateFlags

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (11)

Device state flags. It can be any reasonable combination of the following.

| Value   | Meaning   |
|---|---|
| DISPLAY_DEVICE_ATTACHED_TO_DESKTOP<br>1 (0x1)       | The device is part of the desktop.  |
| DISPLAY_DEVICE_MIRRORING_DRIVER<br>8 (0x8)          | Represents a pseudo device used to mirror application drawing for connecting to a remote computer or other purposes. An invisible pseudo monitor is associated with this device. For example, NetMeeting uses it. |
| DISPLAY_DEVICE_MODESPRUNED<br>134217728 (0x8000000) | The device has more display modes than its output devices support.  |
| DISPLAY_DEVICE_PRIMARY_DEVICE<br>4 (0x4)            | The primary desktop is on the device. For a system with a single display card, this is always set. For a system with multiple display cards, only one device can have this set.                                   |
| DISPLAY_DEVICE_REMOVABLE<br>32 (0x20)               | The device is removable; it cannot be the primary display.  |
| DISPLAY_DEVICE_VGA_COMPATIBLE<br>16 (0x10)          | The device is VGA compatible.   |

## VRefresh

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (5)

Current refresh rate, in hertz.

### XResolution

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (2)

Current number of horizontal pixels.

### YResolution

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId** (3)

Current number of vertical pixels.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | None supported                          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[SystemConfig\\_V0](#)

# TcpIp class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for TCP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{9a280ac0-c8e0-11d1-84e2-00c04fb998a2"}), EventVersion(2)]
class TcpIp : MSNT_SystemTrace
{
};
```

## Members

The **TcpIp** class does not define any members.

## Remarks

To enable TCP/IP events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_NETWORK\_TCPIP** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function.

Event trace consumers can implement special processing for TCP/IP events by calling the [SetTraceCallback](#) function and specifying **TcpIpGuid** as the *pGuid* parameter. Use the following event types to identify the actual network (TCP/IP) event when consuming events.

| Event type   | Description  |
|--|--|
| <b>EVENT_TRACE_TYPE_ACCEPT</b> (Event type value is 15)  | Accept event for IPv4 protocol. The <a href="#">Tcplp_TypeGroup2</a> MOF class defines the event data for this event.  |
| <b>EVENT_TRACE_TYPE_CONNECT</b> (Event type value is 12) | Connect event for IPv4 protocol. The <a href="#">Tcplp_TypeGroup2</a> MOF class defines the event data for this event. |

| Event type  | Description  |
|---|--|
| EVENT_TRACE_TYPE_DISCONNECT(Event type value is 13) | Disconnect event for IPv4 protocol. The <a href="#">Tcplp_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_RECEIVE(Event type value is 11)    | Receive event for IPv4 protocol. The <a href="#">Tcplp_TypeGroup1</a> MOF class defines the event data for this event.   |
| EVENT_TRACE_TYPE_RECONNECT(Event type value is 16)  | Reconnect event for IPv4 protocol. (A connect attempt failed and another attempt is made.) The <a href="#">Tcplp_TypeGroup1</a> MOF class defines the event data for this event. |
| EVENT_TRACE_TYPE_RETRANSMIT(Event type value is 14) | Retransmit event for IPv4 protocol. The <a href="#">Tcplp_TypeGroup1</a> MOF class defines the event data for this event.  |
| EVENT_TRACE_TYPE_SEND(Event type value is 10)       | Send event for IPv4 protocol. The <a href="#">Tcplp_SendIPV4</a> MOF class defines the event data for this event.  |
| Event type value, 17                                | Fail event. The <a href="#">Tcplp_Fail</a> MOF class defines the event data for this event.  |
| Event type value, 18                                | TCP copy event for IPv4 protocol. The <a href="#">Tcplp_TypeGroup1</a> MOF class defines the event data for this event.  |
| Event type value, 26                                | Send event for IPv6 protocol. The <a href="#">Tcplp_SendIPV6</a> MOF class defines the event data for this event.  |
| Event type value, 27                                | Receive event for IPv6 protocol. The <a href="#">Tcplp_TypeGroup3</a> MOF class defines the event data for this event.   |
| Event type value, 28                                | Connect event for IPv6 protocol. The <a href="#">Tcplp_TypeGroup4</a> MOF class defines the event data for this event.   |
| Event type value, 29                                | Disconnect event for IPv6 protocol. The <a href="#">Tcplp_TypeGroup3</a> MOF class defines the event data for this event.  |
| Event type value, 30                                | Retransmit event for IPv6 protocol. The <a href="#">Tcplp_TypeGroup3</a> MOF class defines the event data for this event.  |
| Event type value, 31                                | Accept event for IPv6 protocol. The <a href="#">Tcplp_TypeGroup4</a> MOF class defines the event data for this event.  |

| Event type           | Description  |
|----------------------|--|
| Event type value, 32 | Reconnect event for IPv6 protocol. (A connect attempt failed and another attempt is made.) The <a href="#">Tcplp_TypeGroup3</a> MOF class defines the event data for this event. |
| Event type value, 34 | TCP copy event for IPv6 protocol. The <a href="#">Tcplp_TypeGroup3</a> MOF class defines the event data for this event.  |

You can trace network events to a source and destination process using the **ProcessId** property. Because some network events are logged by separate threads, you may not be able to use the **ProcessId** and **ThreadId** members of [EVENT\\_TRACE\\_HEADER](#) to identify the process or thread that originated the network activities.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Tcplp\\_Fail](#)

[Tcplp\\_SendIPV4](#)

[Tcplp\\_SendIPV6](#)

[Tcplp\\_TypeGroup1](#)

[Tcplp\\_TypeGroup2](#)

[Tcplp\\_TypeGroup3](#)

[Tcplp\\_TypeGroup4](#)

[Tcplp\\_V0](#)

TcpIp\_V1

# TcpIp\_Fail class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for TCP/IP failure events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{17}, EventTypeName{"Fail"}]
class TcpIp_Fail : TcpIp
{
    uint16 Proto;
    uint16 FailureCode;
};
```

## Members

The **TcpIp\_Fail** class has these types of members:

- [Properties](#)

## Properties

The **TcpIp\_Fail** class has these properties.

### FailureCode

Data type: **uint16**

Access type: Read-only

Reason for the failure. Can be one of the following codes:

**ERROR\_INSUFFICIENT\_RESOURCES** (1)

**ERROR\_TOO\_MANY\_ADDRESSES** (2)

**ERROR\_ADDRESS\_EXISTS** (3)

**ERROR\_INVALID\_ADDRESS** (4)

**ERROR\_OTHER** (5)

**ERROR\_TIMEWAIT\_ADDRESS\_EXIST** (6)

### Proto

Data type: **uint16**

Access type: Read-only

Identifies the protocol. Can be one of the following values:

| Value                 | Meaning   |
|-----------------------|---|
| <b>AF_INET</b><br>2   | The Internet Protocol version 4 (IPv4) address family.  |
| <b>AF_INET6</b><br>23 | The Internet Protocol version 6 (IPv6) address family.. |

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Tcplp](#)

# TcpIp\_SendIPV4 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for IPv4 TCP/IP send events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10}, EventTypeName{"SendIPV4"}]
class TcpIp_SendIPV4 : TcpIp
{
    uint32 PID;
    uint32 size;
    object daddr;
    object saddr;
    object dport;
    object sport;
    uint32 starttime;
    uint32 endtime;
    uint32 seqnum;
    uint32 connid;
};
```

## Members

The **TcpIp\_SendIPV4** class has these types of members:

- [Properties](#)

## Properties

The **TcpIp\_SendIPV4** class has these properties.

connid

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(10), Pointer

A unique connection identifier to correlate events belonging to the same connection.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("IPAddrV4")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("Port")

Destination port number.

**endtime**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(8)

End send request time.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Identifier of the process associated with the request.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("IPAddrV4")

Source IP address.

seqnum

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(9)

Sequence number.

size

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Source port number.

**starttime**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7)

Start send request time.

## Requirements

| Requirement              | Value                             |
|--------------------------|-----------------------------------|
| Minimum supported client | Windows Vista [desktop apps only] |

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Tcplp](#)

# TcpIp\_SendIPV6 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for IPv6 TCP/IP send events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{26}, EventTypeName{"SendIPV6"}]
class TcpIp_SendIPV6 : TcpIp
{
    uint32 PID;
    uint32 size;
    object daddr;
    object saddr;
    object dport;
    object sport;
    uint32 starttime;
    uint32 endtime;
    uint32 seqnum;
    uint32 connid;
};
```

## Members

The **TcpIp\_SendIPV6** class has these types of members:

- [Properties](#)

## Properties

The **TcpIp\_SendIPV6** class has these properties.

connid

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(10), Pointer

A unique connection identifier to correlate events belonging to the same connection.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("IPAddrV6")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("Port")

Destination port number.

**endtime**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(8)

End send request time.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Identifier of the process associated with the request.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("IPAddrV6")

Source IP address.

seqnum

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(9)

Sequence number.

size

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Source port number.

**starttime**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7)

Start send request time.

## Requirements

| Requirement              | Value                             |
|--------------------------|-----------------------------------|
| Minimum supported client | Windows Vista [desktop apps only] |

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Tcplp](#)

# TcpIp\_TypeGroup1 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for IPv4 TCP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{11, 13, 14, 16, 18}, EventTypeName{"RecvIPV4", "DisconnectIPV4",  
"RetransmitIPV4", "ReconnectIPV4", "TCPCopyIPV4"}]  
class TcpIp_TypeGroup1 : TcpIp  
{  
    uint32 PID;  
    uint32 size;  
    object daddr;  
    object saddr;  
    object dport;  
    object sport;  
    uint32 seqnum;  
    uint32 connid;  
};
```

## Members

The **TcpIp\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **TcpIp\_TypeGroup1** class has these properties.

connid

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(8)**, **Pointer**

A unique connection identifier to correlate events belonging to the same connection.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("IPAddrV4")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("Port")

Destination port number.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Identifier of the process associated with the request.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("IPAddrV4")

Source IP address.

seqnum

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7)

Sequence number.

size

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Source port number.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Tcplp](#)

# TcpIp\_TypeGroup2 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for IPv4 TCP/IP connect and accept events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{12, 15}, EventTypeName{"ConnectIPV4", "AcceptIPV4"}]
class TcpIp_TypeGroup2 : TcpIp
{
    uint32 PID;
    uint32 size;
    object daddr;
    object saddr;
    object dport;
    object sport;
    uint16 mss;
    uint16 sackopt;
    uint16 tsopt;
    uint16 wsopt;
    uint32 rcvwin;
    sint16 rcvwinscale;
    sint16 sndwinscale;
    uint32 seqnum;
    uint32 connid;
};
```

## Members

The **TcpIp\_TypeGroup2** class has these types of members:

- [Properties](#)

## Properties

The **TcpIp\_TypeGroup2** class has these properties.

**connid**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(15)**, **Pointer**

A unique connection identifier to correlate events belonging to the same connection.

### **daddr**

Data type: **object**

Access type: Read-only

Qualifiers: **WmiDataId(3)**, **Extension("IPAddrV4")**

Destination IP address.

### **dport**

Data type: **object**

Access type: Read-only

Qualifiers: **WmiDataId(5)**, **Extension("Port")**

Destination port number.

### **mss**

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId(7)**

Maximum segment size.

### **PID**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(1)**

Identifier of the process associated with the request.

### **rcvwin**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(11)**

TCP Receive Window size.

### **rcvwinscale**

Data type: **sint16**

Access type: Read-only

Qualifiers: **WmiDataId(12)**

TCP Receive Window Scaling factor.

### **sackopt**

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId(8)**

Selective Acknowledgment (SACK) option in TCP header.

### **saddr**

Data type: **object**

Access type: Read-only

Qualifiers: **WmiDataId(4), Extension("IPAddrV4")**

Source IP address.

### **seqnum**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(14)**

Sequence number.

### **size**

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(2)**

Size of the packet.

### **sdwinscale**

Data type: **sint16**

Access type: Read-only

Qualifiers: **WmiDataId(13)**

TCP Send Window Scaling factor.

### **sport**

Data type: **object**

Access type: Read-only

Qualifiers: **WmiDataId(6), Extension("Port")**

Source port number.

### **tsopt**

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId(9)**

Time Stamp option in TCP header.

### **wsopt**

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId(10)**

Window Scale option in TCP header.

## **Requirements**

---

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Tcplp](#)

# TcpIp\_TypeGroup3 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for IPv6 TCP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{27, 29, 30, 32, 34}, EventTypeName{"RecvIPV6", "DisconnectIPV6",  
"RetransmitIPV6", "ReconnectIPV6", "TCPCopyIPV6"}]  
class TcpIp_TypeGroup3 : TcpIp  
{  
    uint32 PID;  
    uint32 size;  
    object daddr;  
    object saddr;  
    object dport;  
    object sport;  
    uint32 seqnum;  
    uint32 connid;  
};
```

## Members

The **TcpIp\_TypeGroup3** class has these types of members:

- [Properties](#)

## Properties

The **TcpIp\_TypeGroup3** class has these properties.

connid

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(6)**, **Pointer**

A unique connection identifier to correlate events belonging to the same connection.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("IPAddrV6")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("Port")

Destination port number.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Identifier of the process associated with the request.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("IPAddrV6")

Source IP address.

seqnum

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7)

Sequence number.

size

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Source port number.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Tcplp](#)

# TcpIp\_TypeGroup4 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for IPv6 TCP/IP connect and accept events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{28, 31}, EventTypeName{"ConnectIPV6", "AcceptIPV6"}]
class TcpIp_TypeGroup4 : TcpIp
{
    uint32 PID;
    uint32 size;
    object daddr;
    object saddr;
    object dport;
    object sport;
    uint16 mss;
    uint16 sackopt;
    uint16 tsopt;
    uint16 wsopt;
    uint32 rcvwin;
    sint16 rcvwinscale;
    sint16 sndwinscale;
    uint32 seqnum;
    uint32 connid;
};
```

## Members

The **TcpIp\_TypeGroup4** class has these types of members:

- [Properties](#)

## Properties

The **TcpIp\_TypeGroup4** class has these properties.

connid

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(15), Pointer

A unique connection identifier to correlate events belonging to the same connection.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("IPAddrV6")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("Port")

Destination port number.

mss

Data type: **uint16**

Access type: Read-only

Qualifiers: WmiDataId(7)

Maximum segment size.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Identifier of the process associated with the request.

rcvwin

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(11)

TCP Receive Window size.

### **rcvwinscale**

Data type: **sint16**

Access type: Read-only

Qualifiers: WmiDataId(12)

TCP Receive Window Scaling factor.

### **sackopt**

Data type: **uint16**

Access type: Read-only

Qualifiers: **WmiDataId** (8)

Selective Acknowledgment (SACK) option in TCP header.

### **saddr**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("IPAddrV6")

Source IP address.

### **seqnum**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(14)

Sequence number.

### **size**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the packet.

### **sdwinscale**

Data type: **sint16**

Access type: Read-only

Qualifiers: WmiDataId(13)

TCP Send Window Scaling factor.

### **sport**

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Source port number.

### **tsopt**

Data type: **uint16**

Access type: Read-only

Qualifiers: WmiDataId(9)

Time Stamp option in TCP header.

### **wsopt**

Data type: **uint16**

Access type: Read-only

Qualifiers: WmiDataId(10)

Window Scale option in TCP header.

## **Requirements**

---

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Tcplp](#)

# TcpIp\_V0 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for TCP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{9a280ac0-c8e0-11d1-84e2-00c04fb998a2"}), EventVersion(0)]
class TcpIp_V0 : MSNT_SystemTrace
{
};
```

## Members

The **TcpIp\_V0** class does not define any members.

## Requirements

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |
| Minimum supported server | None supported                 |

## See also

[MSNT\\_SystemTrace](#)

[TcpIp](#)

[TcpIp\\_V0\\_TypeGroup1](#)

[TcpIp\\_V1](#)

# TcpIp\_V0\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for TCP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11, 12, 13, 14, 15}, EventTypeName{"Send", "Recv", "Connect",
"Disconnect", "Retransmit", "Accept"}]
class TcpIp_V0_TypeGroup1 : TcpIp_V0
{
    object daddr;
    object saddr;
    object dport;
    object sport;
    uint32 size;
    uint32 PID;
};
```

## Members

The `TcpIp_V0_TypeGroup1` class has these types of members:

- [Properties](#)

## Properties

The `TcpIp_V0_TypeGroup1` class has these properties.

`daddr`

Data type: `object`

Access type: Read-only

Qualifiers: `WmiDataId(1)`, `Extension("IPAddr")`

Destination IP address.

`dport`

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("Port")

Destination port number.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6)

Identifier of the process associated with the request.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(2), Extension("IPAddr")

Source IP address.

size

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5)

Size of the packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("Port")

Source port number.

# Requirements

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |
| Minimum supported server | None supported                 |

## See also

[Tcplp\\_V0](#)

# TcpIp\_V1 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for TCP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{9a280ac0-c8e0-11d1-84e2-00c04fb998a2"}), EventVersion(1)]
class TcpIp_V1 : MSNT_SystemTrace
{
};
```

## Members

The **TcpIp\_V1** class does not define any members.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[TcpIp](#)

[TcpIp\\_V0](#)

[TcpIp\\_V1\\_TypeGroup1](#)

# TcpIp\_V1\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for TCP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11, 12, 13, 14, 15, 16}, EventTypeName{"Send", "Recv",
"Connect", "Disconnect", "Retransmit", "Accept", "Reconnect"}]
class TcpIp_V1_TypeGroup1 : TcpIp_V1
{
    uint32 PID;
    uint32 size;
    object daddr;
    object saddr;
    object dport;
    object sport;
};
```

## Members

The **TcpIp\_V1\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **TcpIp\_V1\_TypeGroup1** class has these properties.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("IPAddr")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("Port")

Destination port number.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Identifier of the process associated with the request.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("IPAddr")

Source IP address.

size

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Source port number.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[Tcplp\\_V1](#)

# Thread class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for thread events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{3d6fa8d1-fe05-11d0-9dda-00c04fd7ba7c}"), EventVersion(3)]
class Thread : MSNT_SystemTrace
{
};
```

## Members

The **Thread** class does not define any members.

## Remarks

To enable thread events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_THREAD** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function.

Event trace consumers can implement special processing for thread events by calling the [SetTraceCallback](#) function and specifying **ThreadId** as the *pGuid* parameter. Use the following event types to identify the actual thread event when consuming events.

| Event type  | Description   |
|---|---|
| <b>EVENT_TRACE_TYPE_END</b> (Event type value is 2)   | End thread event. The <a href="#">Thread_TypeGroup1</a> MOF class defines the event data for this event.  |
| <b>EVENT_TRACE_TYPE_START</b> (Event type value is 1) | Start thread event. The <a href="#">Thread_TypeGroup1</a> MOF class defines the event data for this event.  |
| Event type value, 3                                   | Start data collection thread event. Enumerates threads that are currently running at the time the kernel session starts. The <a href="#">Thread_TypeGroup1</a> MOF class defines the event data for this event. |

| Event type          | Description   |
|---------------------|---|
| Event type value, 4 | End data collection thread event. Enumerates threads that are currently running at the time the kernel session ends. The <a href="#">Thread_TypeGroup1</a> MOF class defines the event data for this event. |

Process and thread start events may be logged in the context of the parent process or thread. As a result, the **ProcessId** and **ThreadId** members of [EVENT\\_TRACE\\_HEADER](#) may not correspond to the process and thread being created. This is why these events contain the process and thread identifiers in the event data (in addition to those in the event header).

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Thread\\_TypeGroup1](#)

[Thread\\_V0](#)

[Thread\\_V1](#)

[Thread\\_V2](#)

# Thread\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for thread start and end events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{1, 2, 3, 4}, EventTypeName{"Start", "End", "DCStart", "DCEnd"}]
class Thread_TypeGroup1 : Thread
{
    uint32 ProcessId;
    uint32 TThreadId;
    uint32 StackBase;
    uint32 StackLimit;
    uint32 UserStackBase;
    uint32 UserStackLimit;
    uint32 Affinity;
    uint32 Win32StartAddr;
    uint32 TebBase;
    uint32 SubProcessTag;
    uint8 BasePriority;
    uint8 PagePriority;
    uint8 IoPriority;
    uint8 ThreadFlags;
};
```

## Members

The **Thread\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Thread\_TypeGroup1** class has these properties.

Affinity

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7), Pointer

The set of processors on which the thread is allowed to run.

BasePriority

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(11)

The scheduler priority of the thread (see the [SetThreadPriority](#) function).

IoPriority

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(13)

An IO priority hint for scheduling IOs generated by the thread.

PagePriority

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(12)

A memory page priority hint for memory pages accessed by the thread.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Format("x")

Process identifier of the thread involved in the event.

StackBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Base address of the thread's stack.

StackLimit

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Pointer

Limit of the thread's stack.

SubProcessTag

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(10), Format("x")

Identifies the service if the thread is owned by a service; otherwise, zero.

TebBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(9), Pointer

Thread environment block base address.

ThreadFlags

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(14)

Not used.

TThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Format("x")

Thread identifier of the thread involved in the event.

UserStackBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5), Pointer

Base address of the thread's user-mode stack.

UserStackLimit

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6), Pointer

Limit of the thread's user-mode stack.

Win32StartAddr

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(8), Pointer

Starting address of the function to be executed by this thread.

## Remarks

The DCStart and DCEnd event types enumerate the threads that are currently running at the time the kernel session starts and ends, respectively.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Thread](#)

# Thread\_V0 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for thread events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{3d6fa8d1-fe05-11d0-9dda-00c04fd7ba7c}"), EventVersion(0)]
class Thread_V0 : MSNT_SystemTrace
{
};
```

## Members

The **Thread\_V0** class does not define any members.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[MSNT\\_SystemTrace](#)

[Thread](#)

[Thread\\_V0\\_TypeGroup1](#)

[Thread\\_V1](#)

# Thread\_V0\_TypeGroup1 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for thread events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{1, 2, 3, 4}, EventTypeName{"Start", "End", "DCStart", "DCEnd"}]
class Thread_V0_TypeGroup1 : Thread_V0
{
    uint32 TThreadId;
    uint32 ProcessId;
};
```

## Members

The **Thread\_V0\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Thread\_V0\_TypeGroup1** class has these properties.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Format("x")

Process identifier of the thread involved in the event.

TThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Format("x")

Thread identifier of the thread involved in the event.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[Thread\\_V0](#)

# Thread\_V1 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for thread events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{3d6fa8d1-fe05-11d0-9dda-00c04fd7ba7c}"), EventVersion(1)]
class Thread_V1 : MSNT_SystemTrace
{
};
```

## Members

The **Thread\_V1** class does not define any members.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Thread](#)

[Thread\\_V0](#)

[Thread\\_V1\\_TypeGroup1](#)

[Thread\\_V1\\_TypeGroup2](#)



# Thread\_V1\_TypeGroup1 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for thread start events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{1, 3}, EventTypeName{"Start", "DCStart"}]
class Thread_V1_TypeGroup1 : Thread_V1
{
    uint32 ProcessId;
    uint32 TThreadId;
    uint32 StackBase;
    uint32 StackLimit;
    uint32 UserStackBase;
    uint32 UserStackLimit;
    uint32 StartAddr;
    uint32 Win32StartAddr;
    sint8 WaitMode;
};
```

## Members

The **Thread\_V1\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Thread\_V1\_TypeGroup1** class has these properties.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(1)**

Process identifier of the thread involved in the event.

**Windows Server 2003:** Contains the Format("x") qualifier.

StackBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Base address of the thread's stack.

StackLimit

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Pointer

Limit of the thread's stack.

StartAddr

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7), Pointer

Memory address at which the trace starts.

TThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Thread identifier of the thread involved in the event.

**Windows Server 2003:** Contains the Format("x") qualifier.

UserStackBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5), Pointer

Base address of the thread's user-mode stack.

UserStackLimit

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6), Pointer

Limit of the thread's user-mode stack.

WaitMode

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(9), Format("c")

Not used.

Win32StartAddr

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(8), Pointer

Starting address of the function to be executed by this thread.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[Thread\\_V1](#)



# Thread\_V1\_TypeGroup2 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for thread end events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{2, 4}, EventTypeName{"End", "DCEnd"}]
class Thread_V1_TypeGroup2 : Thread_V1
{
    uint32 ProcessId;
    uint32 TThreadId;
};
```

## Members

The **Thread\_V1\_TypeGroup2** class has these types of members:

- [Properties](#)

## Properties

The **Thread\_V1\_TypeGroup2** class has these properties.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Process identifier of the thread involved in the event.

**Windows Server 2003:** Contains the Format("x") qualifier.

TThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Thread identifier of the thread involved in the event.

**Windows Server 2003:** Contains the Format("x") qualifier.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[Thread\\_V1](#)

# Thread\_V2 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for thread events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{3d6fa8d1-fe05-11d0-9dda-00c04fd7ba7c}"), EventVersion(2)]
class Thread_V2 : MSNT_SystemTrace
{
};
```

## Members

The **Thread\_V2** class does not define any members.

## Remarks

To enable thread events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_THREAD** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the **StartTrace** function. You can also specify the following flags:

- **EVENT\_TRACE\_FLAG\_CSWITCH**
- **EVENT\_TRACE\_FLAG\_DISPATCHER**

Event trace consumers can implement special processing for thread events by calling the **SetTraceCallback** function and specifying **ThreadId** as the *pGuid* parameter. Use the following event types to identify the actual thread event when consuming events.

| Event type  | Description  |
|---|--|
| <b>EVENT_TRACE_TYPE_END</b> (Event type value is 2)   | End thread event. The <b>Thread_V2_TypeGroup1</b> MOF class defines the event data for this event.   |
| <b>EVENT_TRACE_TYPE_START</b> (Event type value is 1) | Start thread event. The <b>Thread_V2_TypeGroup1</b> MOF class defines the event data for this event. |

| Event type           | Description  |
|----------------------|--|
| Event type value, 3  | Start data collection thread event. Enumerates threads that are currently running at the time the kernel session starts. The <a href="#">Thread_V2_TypeGroup1</a> MOF class defines the event data for this event. |
| Event type value, 4  | End data collection thread event. Enumerates threads that are currently running at the time the kernel session ends. The <a href="#">Thread_V2_TypeGroup1</a> MOF class defines the event data for this event.     |
| Event type value, 36 | Context switch event. The <a href="#">CSwitch</a> MOF class defines the event data for this event.   |
| Event type value, 50 | Ready thread event. The <a href="#">ReadyThread</a> MOF class defines the event data for this event.   |

Process and thread start events may be logged in the context of the parent process or thread. As a result, the [ProcessId](#) and [ThreadId](#) members of [EVENT\\_TRACE\\_HEADER](#) may not correspond to the process and thread being created. This is why these events contain the process and thread identifiers in the event data (in addition to those in the event header).

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[CSwitch](#)

[Thread](#)

[Thread\\_TypeGroup1](#)

[Thread\\_V0](#)

[Thread\\_V1](#)

# CSwitch class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for context switch events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{36}, EventTypeName{"CSwitch"}]
class CSwitch : Thread_V2
{
    uint32 NewThreadId;
    uint32 OldThreadId;
    sint8 NewThreadPriority;
    sint8 OldThreadPriority;
    uint8 PreviousCState;
    sint8 SpareByte;
    sint8 OldThreadWaitReason;
    sint8 OldThreadWaitMode;
    sint8 OldThreadState;
    sint8 OldThreadWaitIdealProcessor;
    uint32 NewThreadWaitTime;
    uint32 Reserved;
};
```

## Members

The **CSwitch** class has these types of members:

- [Properties](#)

## Properties

The **CSwitch** class has these properties.

### NewThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Format("x")

New thread ID after the switch.

### **NewThreadPriority**

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(3)

Thread priority of the new thread.

### **NewThreadWaitTime**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(11), Format("x")

Wait time for the new thread.

### **OldThreadId**

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Format("x")

Previous thread ID.

### **OldThreadPriority**

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(4)

Thread priority of the previous thread.

### **OldThreadState**

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(9)

State of the previous thread. The following are the possible state values:

| State | Description                                   |
|-------|---|
| 0     | Initialized                                   |
| 1     | Ready   |
| 2     | Running                                       |
| 3     | Standby                                       |
| 4     | Terminated                                    |
| 5     | Waiting                                       |
| 6     | Transition                                    |
| 7     | DeferredReady (added for Windows Server 2003) |

### **OldThreadWaitIdealProcessor**

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(10), Format("x")

Ideal wait time of the previous thread.

### **OldThreadWaitMode**

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(8)

Wait mode for the previous thread. The following are the possible values:

| State | Description |
|-------|-------------|
| 0     | KernelMode  |
| 1     | UserMode    |

### **OldThreadWaitReason**

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(7)

Wait reason for the previous thread. The following are the possible values:

| <b>State</b> | <b>Description</b> |
|--------------|--------------------|
| 0            | Executive          |
| 1            | FreePage           |
| 2            | PageIn             |
| 3            | PoolAllocation     |
| 4            | DelayExecution     |
| 5            | Suspended          |
| 6            | UserRequest        |
| 7            | WrExecutive        |
| 8            | WrFreePage         |
| 9            | WrPageIn           |
| 10           | WrPoolAllocation   |
| 11           | WrDelayExecution   |
| 12           | WrSuspended        |
| 13           | WrUserRequest      |
| 14           | WrEventPair        |
| 15           | WrQueue            |
| 16           | WrLpcReceive       |
| 17           | WrLpcReply         |
| 18           | WrVirtualMemory    |
| 19           | WrPageOut          |
| 20           | WrRendezvous       |
| 21           | WrKeyedEvent       |
| 22           | WrTerminated       |

| <b>State</b> | <b>Description</b> |
|--------------|--------------------|
| 23           | WrProcessInSwap    |
| 24           | WrCpuRateControl   |
| 25           | WrCalloutStack     |
| 26           | WrKernel           |
| 27           | WrResource         |
| 28           | WrPushLock         |
| 29           | WrMutex            |
| 30           | WrQuantumEnd       |
| 31           | WrDispatchInt      |
| 32           | WrPreempted        |
| 33           | WrYieldExecution   |
| 34           | WrFastMutex        |
| 35           | WrGuardedMutex     |
| 36           | WrRundown          |
| 37           | MaximumWaitReason  |

## PreviousCState

Data type: **uint8**

Access type: Read-only

Qualifiers: WmiDataId(5)

The index of the C-state that was last used by the processor. A value of 0 represents the lightest idle state with higher values representing deeper C-states.

## Reserved

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(12)

Reserved.

## SpareByte

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(6)

Not used.

## Remarks

These events produce a high volume of events.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Thread](#)

[Thread\\_V2](#)

# Thread\_V2\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for thread start and end events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{1, 2, 3, 4}, EventTypeName{"Start", "End", "DCStart", "DCEnd"}]
class Thread_V2_TypeGroup1 : Thread_V2
{
    uint32 ProcessId;
    uint32 TThreadId;
    uint32 StackBase;
    uint32 StackLimit;
    uint32 UserStackBase;
    uint32 UserStackLimit;
    uint32 StartAddr;
    uint32 Win32StartAddr;
    uint32 TebBase;
    uint32 SubProcessTag;
};
```

## Members

The **Thread\_V2\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **Thread\_V2\_TypeGroup1** class has these properties.

ProcessId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Format("x")

Process identifier of the thread involved in the event.

StackBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(3), Pointer

Base address of the thread's stack.

StackLimit

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(4), Pointer

Limit of the thread's stack.

StartAddr

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7), Pointer

Memory address at which the trace starts.

SubProcessTag

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(10), Format("x")

Identifies the service if the thread is owned by a service; otherwise, zero.

TebBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(9), Pointer

Thread environment block base address.

TThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2), Format("x")

Thread identifier of the thread involved in the event.

UserStackBase

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(5), Pointer

Base address of the thread's user-mode stack.

UserStackLimit

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(6), Pointer

Limit of the thread's user-mode stack.

Win32StartAddr

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(8), Pointer

Starting address of the function to be executed by this thread.

## Remarks

The DCStart and DCEnd event types enumerate the threads that are currently running at the time the kernel session starts and ends, respectively.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Thread](#)

[Thread\\_V2](#)

# ReadyThread class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for ready thread events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{50}, EventTypeName{"ReadyThread"}]
class ReadyThread : Thread_V2
{
    uint32 TThreadId;
    sint8 AdjustReason;
    sint8 AdjustIncrement;
    sint8 Flag;
    sint8 Reserved;
};
```

## Members

The ReadyThread class has these types of members:

- [Properties](#)

## Properties

The ReadyThread class has these properties.

AdjustIncrement

Data type: `sint8`

Access type: Read-only

Qualifiers: WmiDataId(3)

The value by which the priority is being adjusted.

AdjustReason

Data type: `sint8`

Access type: Read-only

Qualifiers: WmiDataId(2)

The reason for the priority boost.

| Value | Meaning  |
|-------|--|
| 0     | Ignore the increment.  |
| 1     | Apply the increment, which will decay incrementally at the end of each quantum.                              |
| 2     | Apply the increment as a boost that will decay in its entirety at quantum (typically for priority donation). |

Flag

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(4)

The following are the possible state flags:

| Value | Meaning   |
|-------|---|
| 0x1   | The thread has been readied from DPC (deferred procedure call). |
| 0x2   | The kernel stack is currently swapped out.                      |
| 0x4   | The process address space is swapped out.                       |

Note that when the kernel stack or the process address space is swapped out, there will be an additional ReadyThread event after the kernel stack or the process address space has been swapped back in and the thread is made ready to be dispatched.

Reserved

Data type: **sint8**

Access type: Read-only

Qualifiers: WmiDataId(5)

Reserved.

TThreadId

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1), Format("x")

The thread identifier of the thread being readied for execution.

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only]       |

## See also

[Thread\\_V2](#)

# UdpIp class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for UDP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{bf3a50c5-a9c9-4988-a005-2df0b7c80f80}"), EventVersion(2)]
class UdpIp : MSNT_SystemTrace
{
};
```

## Members

The **UdpIp** class does not define any members.

## Remarks

To enable UDP/IP events in an NT Kernel logging session, specify the **EVENT\_TRACE\_FLAG\_NETWORK\_TCPIP** flag in the **EnableFlags** member of an **EVENT\_TRACE\_PROPERTIES** structure when calling the [StartTrace](#) function.

Event trace consumers can implement special processing for UDP/IP events by calling the [SetTraceCallback](#) function and specifying **UdpIpGuid** as the *pGuid* parameter. Use the following event types to identify the actual network (UDP/IP) event when consuming events.

| Event type   | Description  |
|--|--|
| <b>EVENT_TRACE_TYPE_RECEIVE</b> (Event type value is 11) | Receive event for IPv4 protocol. The <a href="#">UdpIp_TypeGroup1</a> MOF class defines the event data for this event. |
| <b>EVENT_TRACE_TYPE_SEND</b> (Event type value is 10)    | Send event for IPv4 protocol. The <a href="#">UdpIp_TypeGroup1</a> MOF class defines the event data for this event.    |
| Event type value, 17                                     | Fail event. The <a href="#">UdpIp_Fail</a> MOF class defines the event data for this event.                            |

| Event type           | Description  |
|----------------------|--|
| Event type value, 26 | Send event for IPv6 protocol. The <a href="#">Udplp_TypeGroup2</a> MOF class defines the event data for this event.    |
| Event type value, 27 | Receive event for IPv6 protocol. The <a href="#">Udplp_TypeGroup2</a> MOF class defines the event data for this event. |

You can trace network events to a source and destination process using the **ProcessId** property. Because some network events are logged by separate threads, you may not be able to use the **ProcessId** and **ThreadId** members of [EVENT\\_TRACE\\_HEADER](#) to identify the process or thread that originated the network activities.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[Udplp\\_Fail](#)

[Udplp\\_TypeGroup1](#)

[Udplp\\_TypeGroup2](#)

[Udplp\\_V0](#)

[Udplp\\_V1](#)

# UdpIp\_Fail class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for TCP/IP failure events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{17}, EventTypeName{"Fail"}]
class UdpIp_Fail : UdpIp
{
    uint16 Proto;
    uint16 FailureCode;
};
```

## Members

The **UdpIp\_Fail** class has these types of members:

- [Properties](#)

## Properties

The **UdpIp\_Fail** class has these properties.

### FailureCode

Data type: **uint16**

Access type: Read-only

Reason for the failure. Can be one of the following codes:

**ERROR\_INSUFFICIENT\_RESOURCES** (1)

**ERROR\_TOO\_MANY\_ADDRESSES** (2)

**ERROR\_ADDRESS\_EXISTS** (3)

**ERROR\_INVALID\_ADDRESS** (4)

**ERROR\_OTHER** (5)

**ERROR\_TIMEWAIT\_ADDRESS\_EXIST** (6)

### Proto

Data type: **uint16**

Access type: Read-only

Identifies the protocol. Can be one of the following values:

| Value                 | Meaning   |
|-----------------------|---|
| <b>AF_INET</b><br>2   | The Internet Protocol version 4 (IPv4) address family.  |
| <b>AF_INET6</b><br>23 | The Internet Protocol version 6 (IPv6) address family.. |

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[UdpIp](#)

# UdpIp\_TypeGroup1 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for UDP/IP IPv4 send and receive events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11}, EventTypeName{"SendIPV4", "RecvIPV4"}]
class UdpIp_TypeGroup1 : UdpIp
{
    uint32 PID;
    uint32 size;
    object daddr;
    object saddr;
    object dport;
    object sport;
    uint32 seqnum;
    uint32 connid;
};
```

## Members

The **UdpIp\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **UdpIp\_TypeGroup1** class has these properties.

connid

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(8), Pointer

A unique connection identifier to correlate events belonging to the same connection.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("IPAddrV4")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("Port")

Destination port number.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Identifier of the process associated with the request.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("IPAddrV4")

Source IP address.

seqnum

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7)

Sequence number.

size

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Source port number.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Udplp](#)

# UdpIp\_TypeGroup2 class

Article • 01/07/2021 • 2 minutes to read

This class is the event type class for UDP/IP IPv6 send and receive events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{26, 27}, EventTypeName{"SendIPV6", "RecvIPV6"}]
class UdpIp_TypeGroup2 : UdpIp
{
    uint32 PID;
    uint32 size;
    object daddr;
    object saddr;
    object dport;
    object sport;
    uint32 seqnum;
    uint32 connid;
};
```

## Members

The **UdpIp\_TypeGroup2** class has these types of members:

- [Properties](#)

## Properties

The **UdpIp\_TypeGroup2** class has these properties.

connid

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(8), Pointer

A unique connection identifier to correlate events belonging to the same connection.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("IPAddrV6")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("Port")

Destination port number.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Identifier of the process associated with the request.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("IPAddrV6")

Source IP address.

seqnum

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(7)

Sequence number.

size

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Source port number.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows Vista [desktop apps only]       |
| Minimum supported server | Windows Server 2008 [desktop apps only] |

## See also

[Udplp](#)

# UdpIp\_V0 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for UDP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{bf3a50c5-a9c9-4988-a005-2df0b7c80f80}"), EventVersion(0)]
class UdpIp_V0 : MSNT_SystemTrace
{
};
```

## Members

The **UdpIp\_V0** class does not define any members.

## Requirements

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |
| Minimum supported server | None supported                 |

## See also

[MSNT\\_SystemTrace](#)

[UdpIp](#)

[UdpIp\\_V0\\_TypeGroup1](#)

# UdpIp\_V0\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for UDP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11}, EventTypeName{"Send", "Recv"}]
class UdpIp_V0_TypeGroup1 : UdpIp_V0
{
    uint32 context;
    object saddr;
    object sport;
    uint16 size;
    object daddr;
    object dport;
    uint16 dsiz;
```

```
};
```

## Members

The **UdpIp\_V0\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **UdpIp\_V0\_TypeGroup1** class has these properties.

context

Data type: **uint32**

Access type: Read-only

Qualifiers: **WmiDataId(1)**, pointer

Process identifier for the Address Object that made or received the request.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("IPAddr")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Destination port number.

dsize

Data type: **uint16**

Access type: Read-only

Qualifiers: WmiDataId(7)

Size of the destination packet.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(2), Extension("IPAddr")

Source IP address.

size

Data type: **uint16**

Access type: Read-only

Qualifiers: WmiDataId(4)

Size of the source packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("Port")

Source port number.

## Requirements

| Requirement              | Value                          |
|--------------------------|--------------------------------|
| Minimum supported client | Windows XP [desktop apps only] |
| Minimum supported server | None supported                 |

## See also

[UdpIp\\_V0](#)

# UdpIp\_V1 class

Article • 04/27/2021 • 2 minutes to read

This class is the parent class for UDP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[Guid("{bf3a50c5-a9c9-4988-a005-2df0b7c80f80}")]
class UdpIp_V1 : MSNT_SystemTrace
{
};
```

## Members

The **UdpIp\_V1** class does not define any members.

## Remarks

**Windows Server 2003:** Contains the EventVersion(1) class qualifier.

## Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[MSNT\\_SystemTrace](#)

[UdpIp](#)

[UdpIp\\_V0](#)

Udplp\_V1\_TypeGroup1

# UdpIp\_V1\_TypeGroup1 class

Article • 04/27/2021 • 2 minutes to read

This class is the event type class for UDP/IP events.

The following syntax is simplified from MOF code.

## Syntax

### syntax

```
[EventType{10, 11}, EventTypeName{"Send", "Recv"}]
class UdpIp_V1_TypeGroup1 : UdpIp_V1
{
    uint32 PID;
    uint32 size;
    object daddr;
    object saddr;
    object dport;
    object sport;
};
```

## Members

The **UdpIp\_V1\_TypeGroup1** class has these types of members:

- [Properties](#)

## Properties

The **UdpIp\_V1\_TypeGroup1** class has these properties.

daddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(3), Extension("IPAddr")

Destination IP address.

dport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(5), Extension("Port")

Destination port number.

PID

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(1)

Identifier of the process associated with the request.

saddr

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(4), Extension("IPAddr")

Source IP address.

size

Data type: **uint32**

Access type: Read-only

Qualifiers: WmiDataId(2)

Size of the packet.

sport

Data type: **object**

Access type: Read-only

Qualifiers: WmiDataId(6), Extension("Port")

Source port number.

# Requirements

| Requirement              | Value                                   |
|--------------------------|---|
| Minimum supported client | Windows XP [desktop apps only]          |
| Minimum supported server | Windows Server 2003 [desktop apps only] |

## See also

[Udplp\\_V1](#)

[Udplp](#)

# Event Tracing MOF Qualifiers

Article • 08/25/2021 • 9 minutes to read

Use the qualifiers defined in this section when creating your [provider MOF class](#), [event MOF class](#), [event type MOF class](#), and [the properties of the event type MOF class](#). For an example that includes some of these qualifiers, see [Publishing Your Event Schema](#).

## Provider MOF class qualifiers

The following table lists the qualifiers you can specify on a provider MOF class.

| Qualifier | Data type | Description  |
|-----------|-----------|--|
| Guid      | String    | Required. String Guid that uniquely identifies a provider. For example, Guid("{3F92E6E0-9886-434e-85DB-0D11D3904C0A}"). This is the same GUID you use when you call the <a href="#">RegisterTraceGuids</a> function to register your provider. |

## Event MOF class qualifiers

The following table lists the qualifiers you can specify on an event class (the parent class that groups related event type classes).

| Qualifier    | Data type | Description   |
|--------------|-----------|---|
| Guid         | String    | Required. String Guid that identifies a class of events. For example, Guid("{3F92E6E0-9886-434e-85DB-0D11D3904C0A}"). Event providers use the Guid to set the <a href="#">EVENT_TRACE_HEADER.Guid</a> member, so that consumers can determine the class of events they are receiving.   |
| EventVersion | Integer   | This qualifier is optional for the latest version of an event trace class and is required for all older versions of the class. The latest version of the class either does not specify the <b>EventVersion</b> qualifier or has the highest version number. Version numbers begin with 0, for example, EventVersion(0). Typically, when you create a new version of the class, you also rename the previous version to <classname>_Vn, where n is an incremental number starting at 0. For an example, see <a href="#">Filelo</a> and <a href="#">Filelo_V0</a> . |

# Event type MOF class qualifiers

The following table lists the qualifiers you can specify on an event type class (the class that defines the event property data).

| Qualifier     | Value   | Description   |
|---------------|---------|---|
| EventType     | Integer | Required. Identifies the event type class. For example, EventType(1). The event provider uses the same event type value to set <b>EVENT_TRACE_HEADER.Class.Type</b> . If the same MOF class is used for multiple event types (because they use the same event data), specify the event type value as an array of integers, for example, EventType{12,15}.         |
| EventTypeName | String  | Optional. Describes the event type. For example, EventTypeName("Start"). If the same MOF class is used for multiple event types (because they use the same event data), specify the event type name value as an array of strings, for example, EventTypeName{"Start", "End"}. The elements of the EventTypeName array correspond directly to the EventType array. |

# Property qualifiers

The following table lists the qualifiers you can specify on a property.

| Qualifier | Description   |
|-----------|---|
| BitMap    | Specifies the bit positions that map to string values. If you specify this qualifier, you must also specify the <b>BitValues</b> qualifier.   |
| BitValues | String values. If the <b>BitMap</b> qualifier is also specified, the strings correspond directly to the values in the <b>BitMap</b> qualifier. Otherwise, assume the property value is a one-based index into the value strings (bit one corresponds to the first string in the list).  |
| Extension | Provides additional information on how to consume (interpret) the data. The extension value is case-insensitive. Include the value in quotes, for example, Extension("Guid"). Possible extension values are:<br><br>Guid<br>Indicates the property data is a Guid. The MOF data type must be <b>object</b> . The payload is expected to be a <b>GUID</b> structure.<br><br>IPAddr and IPAddrV4<br>The data is an IP V4 address. The MOF data type must be <b>object</b> . The payload is expected to be an unsigned long. Each byte of the unsigned long represents one of the four parts of the IP address (p1.p2.p3.p4). The low- |

| Qualifier | Description  |
|-----------|--|
|           | <p>order byte contains the value for p1, the next byte contains the value for p2, and so on.</p> <p><b>Prior to Windows Vista:</b> The IPAddrV4 extension is not supported.</p>  |
| IPAddrV6  | <p>The data is an IP V6 address. The MOF data type must be <b>object</b>. The payload is expected to be an <b>IN6_ADDR</b> structure.</p> <p><b>Prior to Windows Vista:</b> The IPAddrV6 extension is not supported.</p>   |
| NoPrint   | Indicates that the consumer should not print this data.  |
| Port      | <p>The data identifies a port number. The MOF data type must be <b>object</b>. The payload is expected to be an unsigned short.</p>  |
| RString   | Newline characters were replaced with spaces. The payload is expected to be a null-terminated, ANSI string.  |
| RWString  | Newline characters were replaced with spaces. The payload is expected to be a null-terminated, wide-character string.  |
| Sid       | <p>The data represents a binary blob SID. The MOF data type must be <b>object</b>. The SID is of a variable length. The value contained in the first 4-bytes (<b>ULONG</b>) indicates whether the blob contains a SID. If the first 4-bytes (<b>ULONG</b>) of the blob is nonzero, the blob contains a SID. The first part of the blob contains the TOKEN_USER (the structure is aligned on an 8-byte boundary) and the second part contains the SID. To address the SID portion of the blob:</p> <ul style="list-style-type: none"> <li>Set a byte pointer to the beginning of the blob</li> <li>Multiply the pointer size for the event log by 2 and add the product to the byte pointer (the <b>PointerSize</b> member of <b>TRACE_LOGFILE_HEADER</b> contains the pointer size value)</li> </ul> |
|           | You can use the following macro to determine the length of the SID.  |
| SizeT     | <pre>#define SeLengthSid( Sid ) \ (8 + (4 * ((SID *)Sid)-&gt;SubAuthorityCount))</pre> <p>Indicates that the property contains a pointer value. The size of the pointer value depends on the operating system used to log the event; the payload will contain a 4-byte value for 32-bit systems or an 8-byte value for 64-bit systems. The MOF data type must be <b>object</b>.</p> <p>Consumers should ignore the data type and <b>Format</b> qualifier if the property includes the <b>SizeT</b> extension. To determine the size of data to read for the property, use:</p>   |

| Qualifier     | Description   |      |             |   |  |   |  |   |  |   |  |
|---------------|---|------|-------------|---|--|---|--|---|--|---|--|
|               | <ul style="list-style-type: none"> <li>The <b>PointerSize</b> member of <a href="#">TRACE_LOGFILE_HEADER</a></li> <li>The <b>Flags</b> member of <a href="#">EVENT_HEADER</a></li> </ul> <p><b>Prior to Windows Vista:</b> The <b>PointerSize</b> value may not be accurate. For example, on a 64-bit computer, a 32-bit application will log 4-byte pointers; however, the session will set <b>PointerSize</b> to 8.</p> <p><b>Variant</b><br/>The data represents a blob. The first four bytes (uint32) indicate the size of the blob. The MOF data type must be <b>object</b>.</p> <p><b>WmiTime</b><br/>Translates the time stamp to system time. The MOF data type must be <b>object</b>. The payload is expected to be an unsigned 64-bit integer.</p> <p><b>Prior to Windows Vista:</b> Not available.</p>   |      |             |   |  |   |  |   |  |   |  |
| <b>Format</b> | <p>Defines the format of the property data. For example, including Format("w") on a string property indicates the string is a wide string. Possible values are:</p> <table border="1" data-bbox="445 878 1398 1468"> <thead> <tr> <th data-bbox="445 878 557 941">Term</th><th data-bbox="557 878 1398 941">Description</th></tr> </thead> <tbody> <tr> <td data-bbox="445 968 557 1057">c</td><td data-bbox="557 968 1398 1057">Display the property value as an ASCII character. You can use this qualifier with <b>uint8</b> data types.</td></tr> <tr> <td data-bbox="445 1084 557 1219">s</td><td data-bbox="557 1084 1398 1219">Treat the array of characters as a null-terminated string. The string is a wide-character string if the data type is <b>char16</b>; otherwise, the string is an ASCII character string.</td></tr> <tr> <td data-bbox="445 1246 557 1336">w</td><td data-bbox="557 1246 1398 1336">The property value is a wide-character string. You can use this qualifier with <b>string</b> data types.</td></tr> <tr> <td data-bbox="445 1363 557 1453">x</td><td data-bbox="557 1363 1398 1453">Display the property value as a hexadecimal number. You can use this qualifier with 16-, 32-, and 64-bit integer data types.</td></tr> </tbody> </table> | Term | Description | c | Display the property value as an ASCII character. You can use this qualifier with <b>uint8</b> data types. | s | Treat the array of characters as a null-terminated string. The string is a wide-character string if the data type is <b>char16</b> ; otherwise, the string is an ASCII character string. | w | The property value is a wide-character string. You can use this qualifier with <b>string</b> data types. | x | Display the property value as a hexadecimal number. You can use this qualifier with 16-, 32-, and 64-bit integer data types. |
| Term          | Description   |      |             |   |  |   |  |   |  |   |  |
| c             | Display the property value as an ASCII character. You can use this qualifier with <b>uint8</b> data types.  |      |             |   |  |   |  |   |  |   |  |
| s             | Treat the array of characters as a null-terminated string. The string is a wide-character string if the data type is <b>char16</b> ; otherwise, the string is an ASCII character string.  |      |             |   |  |   |  |   |  |   |  |
| w             | The property value is a wide-character string. You can use this qualifier with <b>string</b> data types.  |      |             |   |  |   |  |   |  |   |  |
| x             | Display the property value as a hexadecimal number. You can use this qualifier with 16-, 32-, and 64-bit integer data types.  |      |             |   |  |   |  |   |  |   |  |

| Qualifier         | Description   |
|-------------------|---|
| Pointer           | <p>Indicates that the property contains a pointer value. The size of the pointer value depends on the operating system used to log the event; the payload will contain a 4-byte value for 32-bit systems or an 8-byte value for 64-bit systems. The MOF data type must be <b>object</b>.</p> <p>Consumers should ignore the data type and <b>Format</b> qualifier if the property includes the <b>SizeT</b> extension. To determine the size of data to read for the property, use:</p> <ul style="list-style-type: none"> <li>• The <b>PointerSize</b> member of <a href="#">TRACE_LOGFILE_HEADER</a></li> <li>• The <b>Flags</b> member of <a href="#">EVENT_HEADER</a></li> </ul> <p><b>Prior to Windows Vista:</b> The <b>PointerSize</b> value may not be accurate. For example, on a 64-bit computer, a 32-bit application will log 4-byte pointers; however, the session will set <b>PointerSize</b> to 8.</p> <p>Note that some events use <b>PointerType</b> instead of <b>Pointer</b>; do not use <b>PointerType</b>.</p> |
| StringTermination | <p>Indicates how the string property is terminated. For example, <code>StringTermination("NullTerminated")</code> indicates the string property is null-terminated. Possible values are:</p> <p><b>Counted</b><br/> The length of the string is embedded at the beginning of the string as a <b>USHORT</b> value.</p> <p><b>NotCounted</b><br/> The string is not null-terminated and the length of the string is not embedded at the beginning of the string. In this case, the string should be the last element and occupy all the space to the end of the event data.</p> <p><b>NullTerminated</b><br/> The string is null-terminated. If you do not specify the <b>StringTermination</b> qualifier, the string is assumed to be null-terminated.</p> <p><b>ReverseCounted</b><br/> The length of the string is embedded at the beginning of the string as a <b>USHORT</b> value in big-endian format.</p>  |
| ValueDescriptions | <p>Provides descriptions for each value in the <b>Values</b> qualifier. The <a href="#">TdEnumerateProviderFieldInformation</a> and <a href="#">TdQueryProviderFieldInformation</a> functions return these descriptions when you try to retrieve keyword and level information. The descriptions are optional. If you do not provide the descriptions, the functions return <b>NULL</b>. See <a href="#">Specifying level and enable flags values for a provider</a> for more details.</p>  |

| Qualifier          | Description   |
|--------------------|---|
| <b>ValueMap</b>    | <p>Specifies the integer index or flag values that map to string values. If you specify this qualifier, you must also specify the <b>Values</b> qualifier, and optionally the <b>ValueType</b> qualifier. Note that ETW does not support the WMI option of having strings for value map values.</p> <p>The following example shows how to use the <b>ValueMap</b>, <b>Values</b>, and <b>ValueType</b> qualifiers.</p> <pre data-bbox="444 482 1404 763"> <b>ValueType("flag")</b>, <b>ValueMap {"0x01", "0x02", "0x04", "0x08"}</b>, <b>Values {"ValueMapFlag1", "ValueMapFlag2", "ValueMapFlag4", "ValueMapFlag8"}</b>]</pre> |
| <b>Values</b>      | <p>String values. If the <b>ValueMap</b> qualifier is also specified, the strings correspond directly to the values in the <b>ValueMap</b> qualifier. Otherwise, assume the property value is a zero-based index into the value strings.</p>  |
| <b>ValueType</b>   | <p>Indicates if the <b>ValueMap</b> values are integer index values or bit flag values. If you do not specify this qualifier, integer index values are assumed. To specify that the values are integer index values, use <b>ValueType("index")</b>. To specify that the values are bit flag values, use <b>ValueType("flag")</b>.</p>   |
| <b>WmiDataId</b>   | <p>Each property must contain the <b>WmiDataId</b> qualifier. <b>WmiDataId</b> defines the order in which the consumer reads the event data. The value for <b>WmiDataId</b> starts at 1 and increments for each property in the class. For example, <b>WmiDataId(1)</b>.</p>  |
| <b>XMLFragment</b> | <p>Indicates that the data is in XML format and ready to display without further formatting.</p>  |

## Specifying level and enable flags values for a provider

To document the level and enable flags that a controller would use to enable your provider, include the "Level" and "Flags" properties in your provider MOF class. The Level and Flags property names are case sensitive. The properties must include the **Values** and **ValueMap** qualifiers, which specify the possible level and enable flag values. The **ValueMap** for the enable flag values must be bit (flag) values. The **ValueDescriptions** qualifier is optional but you should use it to provide descriptions for each possible value. The descriptions are used when someone calls the

[TdEnumerateProviderFieldInformation](#) and [TdQueryProviderFieldInformation](#) functions to get the possible level and enable flags (keywords) values for the provider.

The following shows a provider class that specifies the possible level and enable flags values.

syntax

```
[Dynamic,
Description("IIS_Trace") : amended,
guid("{3a2a4e84-4c21-4981-ae10-3fda0d9b0f83"}),
locale("MS\\0x409")]
class IIS_Trace : EventTrace
{
    [Description ("Enable Flags") : amended,
     ValueDescriptions{
        "Allow_tracing_only_selected_requests",
        "IIS_authentication_events",
        "IIS_security_events",
        "IIS_filter_events",
        "IIS_static_file_events",
        "IIS_CGI_events",
        "IIS_compression_events",
        "IIS_cache_events",
        "IIS_request_notifications_events",
        "IIS_module_events",
        "IIS_FastCGI_events"},

    DefineValues{
        "UseUrlFilter",
        "IISAuthentication",
        "IISSecurity",
        "IISFilter",
        "IISStaticFile",
        "IISCGI",
        "IISCompression",
        "IISCache",
        "IISRequestNotification",
        "IISModule",
        "IISFastCGI"},

    Values{
        "UseUrlFilter",
        "IISAuthentication",
        "IISSecurity",
        "IISFilter",
        "IISStaticFile",
        "IISCGI",
        "IISCompression",
        "IISCache",
        "IISRequestNotification",
        "IISModule",
        "IISFastCGI"},

    ValueMap{
        "0x00000001",
```

```

        "0x00000002",
        "0x00000004",
        "0x00000008",
        "0x00000010",
        "0x00000020",
        "0x00000040",
        "0x00000080",
        "0x00000100",
        "0x00000200",
        "0x00001000"}: amended
    ]
    uint32 Flags;

[Description ("Levels") : amended,
 ValueDescriptions{
    "Abnormal exit or termination",
    "Severe errors that need logging",
    "Warnings such as allocation failure",
    "Includes non-error cases",
    "Detailed traces from intermediate steps" } : amended,
 DefineValues{
    "TRACE_LEVEL_FATAL",
    "TRACE_LEVEL_ERROR",
    "TRACE_LEVEL_WARNING"
    "TRACE_LEVEL_INFORMATION",
    "TRACE_LEVEL_VERBOSE" },
 Values{
    "Fatal",
    "Error",
    "Warning",
    "Information",
    "Verbose" },
 ValueMap{
    "0x1",
    "0x2",
    "0x3",
    "0x4",
    "0x5" },
 ValueType("index")
]
uint32 Level;
};

```

# Event Tracing Tools

Article • 01/07/2021 • 2 minutes to read

The following event tracing tools are available for your use.

| Tool     | Description  |
|----------|--|
| Logman   | Manages and schedules performance counter and event trace log collections on local and remote systems. You can use this tool to list the providers that have <a href="#">published the layout of their event data</a> in the root\wmi namespace. For details on using this tool, see the <a href="#">Help and Support Center</a> . |
| Tracefmt | Formats information in a trace log file in human-readable form. For details, see Tools for Software Tracing in the Microsoft Windows Driver Development Kit (DDK). Used by WPP Software Tracing only.  |
| Tracelog | Controls an event trace session. For details, see Tools for Software Tracing in the DDK.   |
| Tracepdb | Creates a trace message format file that the Tracefmt tool uses to convert logged messages to a human-readable form. For details, see Tools for Software Tracing in the DDK. Used by WPP Software Tracing only.  |
| Tracerpt | Processes event trace logs or real-time events from instrumented event trace providers and allows you to generate trace analysis reports and CSV (comma-delimited) files for the events generated. For details, see the <a href="#">Help and Support Center</a> .  |

# Event Tracing Samples

Article • 01/07/2021 • 2 minutes to read

The Platform Software Development Kit (SDK) contains complete event tracing samples. These samples are located in the Samples\WinBase\EventTrace directory. The root of this path is the base installation directory of the PSDK. You can download the PSDK from <https://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5>.

The following table describes the samples contained in the Samples\WinBase\EventTrace directory.

| Sample   | Description  |
|----------|--|
| TraceDmp | An event trace consumer. It decodes the event data using the format information obtained from WMI and outputs the data in a .csv file. |
| TraceDp  | Uses the event trace provider API to provide event trace data to the logger or a consumer.   |
| TraceLog | Uses the event trace controller API to manage logging sessions.  |

For detailed descriptions, see the Readme.txt file included with these samples.

# Event Tracing

Article • 01/24/2023 16 minutes to read

Overview of the Event Tracing technology.

To develop Event Tracing, you need these headers:

- [evntcons.h](#)
- [evnprov.h](#)
- [evntrace.h](#)
- [relogger.h](#)
- [securitybaseapi.h](#)
- [tdh.h](#)

For programming guidance for this technology, see:

- [Event Tracing](#)

## Enumerations

|  |
|--|
|  |
| <a href="#">_TDH_IN_TYPE</a>   |
| Defines the supported [in] types for a trace data helper (TDH).                      |
| <a href="#">_TDH_OUT_TYPE</a>  |
| Defines the supported [out] types for a trace data helper (TDH).                     |
| <a href="#">DECODING_SOURCE</a>  |
| Defines the source of the event data.  |
| <a href="#">ETW_PROCESS_HANDLE_INFO_TYPE</a>   |
| Specifies the operation that will be performed on a trace processing session.        |
| <a href="#">ETW_PROCESS_TRACE_MODES</a>  |
| Specifies the supported process trace modes.   |
| <a href="#">ETW_PROVIDER_TRAIT_TYPE</a>  |
| Specifies the types of Provider Traits supported by Event Tracing for Windows (ETW). |

## [EVENT\\_FIELD\\_TYPE](#)

Defines the provider information to retrieve.

## [EVENT\\_INFO\\_CLASS](#)

The EVENT\_INFO\_CLASS enumeration type is used with the EventSetInformation function to specify the configuration operation to be performed on an ETW event provider registration.

## [EVENTSECURITYOPERATION](#)

Defines what component of the security descriptor that the EventAccessControl function modifies.

## [MAP\\_FLAGS](#)

Defines constant values that indicate if the map is a value map, bitmap, or pattern map.

## [MAP\\_VALUETYPE](#)

Defines if the value map value is in a ULONG data type or a string.

## [PAYLOAD\\_OPERATOR](#)

Defines the supported payload operators for a trace data helper (TDH).

## [PROPERTY\\_FLAGS](#)

Defines if the property is contained in a structure or array.

## [TDH\\_CONTEXT\\_TYPE](#)

Defines the context type.

## [TEMPLATE\\_FLAGS](#)

Defines constant values that indicates the layout of the event data.

## [TRACE\\_QUERY\\_INFO\\_CLASS](#)

Used with EnumerateTraceGuidsEx and TraceSetInformation to specify a type of trace information.

# Functions

|   |
|---|
|   |
| <a href="#">AddLogFileTraceStream</a>   |
| Adds a new logfile-based ETW trace stream to the relogger.  |
| <a href="#">AddRealtimeTraceStream</a>  |
| Adds a new real-time ETW trace stream to the relogger.  |
| <a href="#">Cancel</a>  |
| Terminates the relogging process.   |
| <a href="#">Clone</a>   |
| Creates a duplicate copy of an event.   |
| <a href="#">CloseTrace</a>  |
| The CloseTrace function closes a trace processing session that was created with OpenTrace.  |
| <a href="#">ControlTraceA</a>   |
| The ControlTraceA (ANSI) function (evntrace.h) flushes, queries, updates, or stops the specified event tracing session.   |
| <a href="#">ControlTraceW</a>   |
| The ControlTraceW (Unicode) function (evntrace.h) flushes, queries, updates, or stops the specified event tracing session.  |
| <a href="#">CreateEventInstance</a>   |
| Generates a new event.  |
| <a href="#">CreateTraceInstanceId</a>   |
| A RegisterTraceGuids-based ("Classic") event provider uses the CreateTraceInstanceId function to create a unique transaction identifier and map it to a registration handle. The provider can then use the transaction identifier when calling the TraceEventInstance function. |
| <a href="#">CveEventWrite</a>   |
| A tracing function for publishing events when an attempted security vulnerability exploit is detected in your user-mode application.  |
| <a href="#">DECLSPEC_XFGVIRT</a>  |

|   |
|---|
|   |
| <b><a href="#">EMI_MAP_FORMAT</a></b>   |
| Macro that retrieves the event map format.  |
| <b><a href="#">EMI_MAP_INPUT</a></b>  |
| Macro that retrieves the event map input.   |
| <b><a href="#">EMI_MAP_NAME</a></b>   |
| Macro that retrieves the event map name.  |
| <b><a href="#">EMI_MAP_OUTPUT</a></b>   |
| Macro that retrieves the event map output.  |
| <b><a href="#">EnableTrace</a></b>  |
| A trace session controller calls EnableTrace to configure how an ETW event provider logs events to a trace session. The EnableTraceEx2 function supersedes this function.   |
| <b><a href="#">EnableTraceEx</a></b>  |
| A trace session controller calls EnableTraceEx to configure how an ETW event provider logs events to a trace session. The EnableTraceEx2 function supersedes this function. |
| <b><a href="#">EnableTraceEx2</a></b>   |
| A trace session controller calls EnableTraceEx2 to configure how an ETW event provider logs events to a trace session.  |
| <b><a href="#">EnumerateTraceGuids</a></b>  |
| Retrieves information about event trace providers that are currently running on the computer. The EnumerateTraceGuidsEx function supersedes this function.                  |
| <b><a href="#">EnumerateTraceGuidsEx</a></b>  |
| Retrieves information about event trace providers that are currently running on the computer.   |
| <b><a href="#">EtwGetTraitFromProviderTraits</a></b>  |
|   |
| <b><a href="#">EventAccessControl</a></b>   |
| Adds or modifies the permissions of the specified provider or session.  |

## [EventAccessQuery](#)

Retrieves the permissions for the specified controller or provider.

## [EventAccessRemove](#)

Removes the permissions defined in the registry for the specified provider or session.

## [EventActivityIdControl](#)

Creates, queries, and sets activity identifiers for use in ETW events.

## [EventDataDescCreate](#)

Sets the values of an EVENT\_DATA\_DESCRIPTOR.

## [EventDescCreate](#)

Sets the values of an event descriptor.

## [EventDescGetChannel](#)

Retrieves the channel from the event descriptor.

## [EventDescGetId](#)

Retrieves the event identifier from the event descriptor.

## [EventDescGetKeyword](#)

Retrieves the keyword from the event descriptor.

## [EventDescGetLevel](#)

Retrieves the severity level from the event descriptor.

## [EventDescGetOpcode](#)

Retrieves the operation code from the event descriptor.

## [EventDescGetTask](#)

Retrieves the task from the event descriptor.

## [EventDescGetVersion](#)

Retrieves the version from the event descriptor.

|   |
|---|
|   |
| <a href="#">EventDescOrKeyword</a>  |
| Adds another keyword to the event descriptor.   |
| <a href="#">EventDescSetChannel</a>   |
| Sets the Channel member of the event descriptor.  |
| <a href="#">EventDescSetId</a>  |
| Sets the Id member of the event descriptor.   |
| <a href="#">EventDescSetKeyword</a>   |
| Sets the Keyword member of the event descriptor.  |
| <a href="#">EventDescSetLevel</a>   |
| Sets the Level member of the event descriptor.  |
| <a href="#">EventDescSetOpcode</a>  |
| Sets the Opcode member of the event descriptor.   |
| <a href="#">EventDescSetTask</a>  |
| Sets the Task member of the event descriptor.   |
| <a href="#">EventDescSetVersion</a>   |
| Sets the Version member of the event descriptor.  |
| <a href="#">EventDescZero</a>   |
| Initializes an event descriptor to zero.  |
| <a href="#">EventEnabled</a>  |
| Determines whether an event provider should generate a particular event based on the event's EVENT_DESCRIPTOR.  |
| <a href="#">EventProviderEnabled</a>  |
| Determines whether an event provider should generate a particular event based on the event's Level and Keyword. |
| <a href="#">EventRegister</a>   |
| Registers an ETW event provider, creating a handle that can be used to write ETW events.                        |

|   |
|---|
|   |
| <p><a href="#">EventSetInformation</a></p> <p>Configures an ETW event provider.</p>   |
| <p><a href="#">EventUnregister</a></p> <p>Unregisters an ETW event provider.</p>  |
| <p><a href="#">EventWrite</a></p> <p>Writes an ETW event that uses the current thread's activity ID.</p>  |
| <p><a href="#">EventWriteEx</a></p> <p>Writes an ETW event with an activity ID, an optional related activity ID, session filters, and special options.</p>  |
| <p><a href="#">EventWriteString</a></p> <p>Writes an ETW event that contains a string as its data. This function should not be used.</p>  |
| <p><a href="#">EventWriteTransfer</a></p> <p>Writes an ETW event with an activity ID and an optional related activity ID.</p>   |
| <p><a href="#">FlushTraceA</a></p> <p>The FlushTraceA (ANSI) function (evntrace.h) causes an event tracing session to immediately deliver buffered events for the specified session.</p>  |
| <p><a href="#">FlushTraceW</a></p> <p>The FlushTraceW (Unicode) function (evntrace.h) causes an event tracing session to immediately deliver buffered events for the specified session.</p>   |
| <p><a href="#">GetEventProcessorIndex</a></p>   |
| <p><a href="#">GetEventRecord</a></p> <p>Retrieves the event record that describes an event.</p>  |
| <p><a href="#">GetTraceEnableFlags</a></p> <p>A RegisterTraceGuids-based ("Classic") event provider uses the GetTraceEnableFlags function to retrieve the enable flags specified by the trace controller to indicate which category of events to trace. Providers call this function from their ControlCallback function.</p> |

## [GetTraceEnableLevel](#)

A RegisterTraceGuids-based ("Classic") event provider uses the GetTraceEnableLevel function to retrieve the enable level specified by the trace controller to indicate which level of events to trace. Providers call this function from their ControlCallback function.

## [GetTraceLoggerHandle](#)

A RegisterTraceGuids-based ("Classic") event provider uses the GetTraceLoggerHandle function to retrieve the handle of the event tracing session to which it should write events. Providers call this function from their ControlCallback function.

## [GetUserContext](#)

Retrieves the user context associated with the stream to which the event belongs.

## [Inject](#)

Injects a non-system-generated event into the event stream being written to the output trace logfile.

## [OnBeginProcessTrace](#)

Indicates that a trace is about to begin so that relogging can be started.

## [OnEvent](#)

Indicates that an event has been received on the trace streams associated with a relogger.

## [OnFinalizeProcessTrace](#)

Indicates that a trace is about to end so that relogging can be finalized.

## [OpenTraceA](#)

The OpenTraceA (ANSI) function (evntrace.h) opens an ETW trace processing handle for consuming events from an ETW real-time trace session or an ETW log file.

## [OpenTraceFromBufferStream](#)

Creates a trace processing session that is not directly attached to any file or active session.

## [OpenTraceFromFile](#)

Creates a trace processing session to process a Tracelog .etl file.

## [OpenTraceFromRealTimeLogger](#)

Opens an ETW trace processing handle for consuming events from an ETW real-time trace session or an ETW log file.

## [OpenTraceFromRealTimeLoggerWithAllocationOptions](#)

Creates a trace processing session attached to an active real-time ETW session.

## [OpenTraceW](#)

The OpenTraceW (Unicode) function (evntrace.h) opens an ETW trace processing handle for consuming events from an ETW real-time trace session or an ETW log file.

## [PEI\\_PROVIDER\\_NAME](#)

Macro that retrieves the Provider Event Info (PEI) name.

## [PENABLECALLBACK](#)

ETW event providers optionally define an EnableCallback function to receive configuration change notifications. The PENABLECALLBACK type defines a pointer to this callback function.

EnableCallback is a placeholder for the application-defined function name.

## [PETW\\_BUFFER\\_CALLBACK](#)

Function definition for the BufferCallback that will be invoked by ProcessTrace.

## [PETW\\_BUFFER\\_COMPLETION\\_CALLBACK](#)

Function definition for the callback that will be fired when ProcessTraceAddBufferToBufferStream is finished with a buffer. This callback should typically free the buffer as appropriate

## [PEVENT\\_CALLBACK](#)

ETW event consumers implement this callback to receive events from a trace processing session. The EventRecordCallback callback supersedes this callback.

## [PEVENT\\_RECORD\\_CALLBACK](#)

ETW event consumers implement this callback to receive events from a trace processing session. The PEVENT\_RECORD\_CALLBACK type defines a pointer to this callback function. EventRecordCallback is a placeholder for the application-defined function name.

## [PEVENT\\_TRACE\\_BUFFER\\_CALLBACKA](#)

The PEVENT\_TRACE\_BUFFER\_CALLBACKA (ANSI) (evntrace.h) function gets statistics about each buffer of events that ETW sends during a trace processing session.

## [PEVENT\\_TRACE\\_BUFFER\\_CALLBACKW](#)

The PEVENT\_TRACE\_BUFFER\_CALLBACKW (Unicode) (evntrace.h) function gets statistics about each buffer of events that ETW sends during a trace processing session.

## [PFI\\_FIELD\\_MESSAGE](#)

Macro that retrieves the Provider Field Information (PFI) field message.

## [PFI\\_FIELD\\_NAME](#)

Macro that retrieves the Provider Field Information (PFI) field name.

## [PFI\\_FILTER\\_MESSAGE](#)

Macro that filters the Provider Field Information (PFI) field message.

## [PFI\\_PROPERTY\\_NAME](#)

Macro that retrieves the Provider Field Information (PFI) property name.

## [ProcessTrace](#)

Delivers events from one or more trace processing sessions to the consumer.

## [ProcessTrace](#)

Delivers events from the associated trace streams to the consumer.

## [ProcessTraceAddBufferToBufferStream](#)

Provides an ETW trace buffer to a processing session created by OpenTraceFromBufferStream.

## [ProcessTraceBufferDecrementReference](#)

Releases a reference to a Buffer that was added by ProcessTraceBufferIncrementReference.

## [ProcessTraceBufferIncrementReference](#)

Called during the BufferCallback on the provided Buffer to prevent it from being freed until the caller is done with it.

## [QueryAllTracesA](#)

The QueryAllTracesA (ANSI) function (evntrace.h) function retrieves the properties and statistics for all event tracing sessions that the caller can query.

## [QueryAllTracesW](#)

The QueryAllTracesW (Unicode) function (evntrace.h) function retrieves the properties and statistics for all event tracing sessions that the caller can query.

## [QueryTraceA](#)

The QueryTraceA (ANSI) function (evntrace.h) retrieves the property settings and session statistics for the specified event tracing session.

## [QueryTraceProcessingHandle](#)

Retrieves information about an ETW trace processing session opened by OpenTrace.

## [QueryTraceW](#)

The QueryTraceW (Unicode) function (evntrace.h) retrieves the property settings and session statistics for the specified event tracing session.

## [RegisterCallback](#)

Registers an implementation of IEventCallback with the relogger in order to signal trace activity (starting, stopping, and logging new events).

## [RegisterTraceGuidsA](#)

The RegisterTraceGuidsA (ANSI) function (evntrace.h) is an obsolete function, and new code should use the provided alternative.

## [RegisterTraceGuidsW](#)

The RegisterTraceGuidsW (Unicode) function (evntrace.h) is an obsolete function, and new code should use the provided alternative.

## [RemoveTraceCallback](#)

The RemoveTraceCallback function stops an EventCallback function from receiving events for an event trace class. This function is obsolete.

## [SetActivityId](#)

Sets the activity ID in the current thread.

## [SetCompressionMode](#)

Enables or disables compression on the relogged trace.

## [SetEventDescriptor](#)

Sets the event descriptor for an event.

## [SetOutputFilename](#)

Indicates the file to which ETW should write the new, relogged trace.

## [SetPayload](#)

Sets the payload for an event.

## [SetProcessId](#)

Assigns an event to a specific process.

## [SetProcessorIndex](#)

Sets the processor index in the current thread.

## [SetProviderId](#)

Sets the GUID for the provider which traced an event.

## [SetThreadId](#)

Sets the identifier of a thread that generates an event.

## [SetThreadTimes](#)

Sets the thread times in the current thread.

## [SetTimeStamp](#)

Sets the time at which an event occurred.

## [SetTraceCallback](#)

The SetTraceCallback function specifies an EventCallback function to process events for the specified event trace class. This function is obsolete.

## [StartTraceA](#)

The StartTrace function starts an event tracing session. (ANSI)

## [StartTraceW](#)

The StartTrace function starts an event tracing session. (Unicode)

## [StopTraceA](#)

The StopTraceA (ANSI) function (`evntrace.h`) stops the specified event tracing session. The `ControlTrace` function supersedes this function.

## [StopTraceW](#)

The StopTraceW (Unicode) function (`evntrace.h`) stops the specified event tracing session. The `ControlTrace` function supersedes this function.

## [TdhAggregatePayloadFilters](#)

Aggregates multiple payload filters for a single provider into a single data structure for use with the `EnableTraceEx2` function.

## [TdhCleanupPayloadEventFilterDescriptor](#)

Frees the aggregated structure of payload filters created using the `TdhAggregatePayloadFilters` function.

## [TdhCloseDecodingHandle](#)

Frees any resources associated with the input decoding handle.

## [TdhCreatePayloadFilter](#)

Creates a single filter for a single payload to be used with the `EnableTraceEx2` function.

## [TdhDeletePayloadFilter](#)

Frees the memory allocated for a single payload filter by the `TdhCreatePayloadFilter` function.

## [TdhEnumerateManifestProviderEvents](#)

Retrieves the list of events present in the provider manifest.

## [TdhEnumerateProviderFieldInformation](#)

Retrieves the specified field metadata for a given provider.

## [TdhEnumerateProviderFilters](#)

Enumerates the filters that the specified provider defined in the manifest.

## [TdhEnumerateProviders](#)

Retrieves a list of providers that have registered a MOF class or manifest file on the computer.

[TdhEnumerateProvidersForDecodingSource](#)

Retrieves a list of providers that have registered a MOF class or manifest file on the computer.

[TdhFormatProperty](#)

Formats a property value for display.

[TdhGetDecodingParameter](#)

Retrieves the value of a decoding parameter.

[TdhGetEventInformation](#)

Retrieves metadata about an event.

[TdhGetEventMapInformation](#)

Retrieves information about the event map contained in the event.

[TdhGetManifestEventInformation](#)

Retrieves metadata about an event in a manifest.

[TdhGetProperty](#)

Retrieves a property value from the event data.

[TdhGetPropertySize](#)

Retrieves the size of one or more property values in the event data.

[TdhGetWppMessage](#)

Retrieves the formatted WPP message embedded into an EVENT\_RECORD structure.

[TdhGetWppProperty](#)

Retrieves a specific property associated with a WPP message.

[TdhLoadManifest](#)

Loads the manifest used to decode a log file.

[TdhLoadManifestFromBinary](#)

Takes a NULL-terminated path to a binary file that contains metadata resources needed to decode a specific event provider.

## [TdhLoadManifestFromMemory](#)

Loads the manifest from memory.

## [TdhOpenDecodingHandle](#)

Opens a decoding handle.

## [TdhQueryProviderFieldInformation](#)

Retrieves information for the specified field from the event descriptions for those field values that match the given value.

## [TdhSetDecodingParameter](#)

Sets the value of a decoding parameter.

## [TdhUnloadManifest](#)

Unloads the manifest that was loaded by the TdhLoadManifest function.

## [TdhUnloadManifestFromMemory](#)

Unloads the manifest from memory.

## [TEI\\_ACTIVITYID\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) activity ID name.

## [TEI\\_CHANNEL\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) channel name.

## [TEI\\_EVENT\\_MESSAGE](#)

Macro that retrieves the Trace Event Information (TEI) message.

## [TEI\\_KEYWORDS\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) keywords name.

## [TEI\\_LEVEL\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) level name.

## [TEI\\_MAP\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) map name.

## [TEI\\_OPCODE\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) opcode name.

## [TEI\\_PROPERTY\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) property name.

## [TEI\\_PROVIDER\\_MESSAGE](#)

Macro that retrieves the Trace Event Information (TEI) provider message.

## [TEI\\_PROVIDER\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) provider name.

## [TEI\\_RELATEDACTIVITYID\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) related activity ID name.

## [TEI\\_TASK\\_NAME](#)

Macro that retrieves the Trace Event Information (TEI) task name.

## [TraceEvent](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceEvent function to send a structured event to an event tracing session.

## [TraceEventInstance](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceEventInstance function to send a structured event to an event tracing session with an instance identifier.

## [TraceMessage](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceMessage function to send a message-based (TMF-based WPP) event to an event tracing session.

## [TraceMessageVa](#)

A RegisterTraceGuids-based ("Classic") event provider uses the TraceMessageVa function to send a message-based (TMF-based WPP) event to an event tracing session using va\_list parameters.

## [TraceQueryInformation](#)

Provides information about an event tracing session.

|  |
|--|
|  |
| <a href="#">TraceSetInformation</a>  |
| Configures event tracing session settings.   |
| <a href="#">UnregisterTraceGuids</a>   |
| Unregisters a "Classic" (Windows 2000-style) ETW event trace provider that was registered using RegisterTraceGuids.  |
| <a href="#">UpdateTraceA</a>   |
| The UpdateTraceA (ANSI) function (evntrace.h) updates the property setting of the specified event tracing session.   |
| <a href="#">UpdateTraceW</a>   |
| The UpdateTraceW (Unicode) function (evntrace.h) updates the property setting of the specified event tracing session.  |
| <a href="#">WMIDPREQUEST</a>   |
| A RegisterTraceGuids-based ("Classic") event provider implements this function to receive notifications from controllers. The WMIDPREQUEST type defines a pointer to this callback function. ControlCallback is a placeholder for the application-defined function name. |

## Interfaces

|   |
|---|
|   |
| <a href="#">ITraceEvent</a>   |
| Provides access to data relating to a specific event.   |
| <a href="#">ITraceEventCallback</a>   |
| Used by ETW to provide information to the relogger as the tracing process starts, ends, and logs events.              |
| <a href="#">ITraceRelogger</a>  |
| Provides access to the relogging functionality, allowing you to manipulate and relog events from an ETW trace stream. |

## Structures

## [CLASSIC\\_EVENT\\_ID](#)

Identifies the kernel event for which you want to enable call stack tracing.

## [ENABLE\\_TRACE\\_PARAMETERS](#)

Contains information used to enable a provider via EnableTraceEx2.

## [ENABLE\\_TRACE\\_PARAMETERS\\_V1](#)

Contains information used to enable a provider via EnableTraceEx2. This structure is obsolete.

## [ETW\\_BUFFER\\_CALLBACK\\_INFORMATION](#)

Provided to the BufferCallback as the *ConsumerInfo* parameter and provides details on the current processing session.

## [ETW\\_BUFFER\\_CONTEXT](#)

Provides context information about the event.

## [ETW\\_BUFFER\\_CONTEXT](#)

Provides context information about the event. (ETW\_BUFFER\_CONTEXT)

## [ETW\\_BUFFER\\_HEADER](#)

The header structure of an ETW buffer.

## [ETW\\_OPEN\\_TRACE\\_OPTIONS](#)

Provides configuration parameters to OpenTraceFromBufferStream, OpenTraceFromFile, OpenTraceFromRealTimeLogger, OpenTraceFromRealTimeLoggerWithAllocationOptions functions.

## [ETW\\_TRACE\\_PARTITION\\_INFORMATION](#)

Contains partition information pulled from an ETW trace.

## [EVENT\\_DATA\\_DESCRIPTOR](#)

The EVENT\_DATA\_DESCRIPTOR structure defines a block of data that will be used in an ETW event.

## [EVENT\\_DESCRIPTOR](#)

The EVENT\_DESCRIPTOR structure contains information (metadata) about an ETW event.

## [EVENT\\_DESCRIPTOR](#)

Contains metadata that defines the event.

|  |
|--|
| EVENT_EXTENDED_ITEM_EVENT_KEY  |
| EVENT_EXTENDED_ITEM_INSTANCE   |
| Defines the relationship between events if TraceEventInstance was used to log related events.        |
| EVENT_EXTENDED_ITEM_PEBS_INDEX   |
| EVENT_EXTENDED_ITEM_PMC_COUNTERS   |
| EVENT_EXTENDED_ITEM_PROCESS_START_KEY  |
| EVENT_EXTENDED_ITEM RELATED_ACTIVITYID   |
| Defines the parent event of this event.  |
| EVENT_EXTENDED_ITEM_STACK_KEY32  |
| EVENT_EXTENDED_ITEM_STACK_KEY64  |
| EVENT_EXTENDED_ITEM_STACK_TRACE32  |
| Defines a call stack on a 32-bit computer.   |
| EVENT_EXTENDED_ITEM_STACK_TRACE64  |
| Defines a call stack on a 64-bit computer.   |
| EVENT_EXTENDED_ITEM_TS_ID  |
| Defines the terminal session that logged the event.  |
| EVENT_FILTER_DESCRIPTOR  |
| Defines the filter data that a session passes to the provider's enable callback function.            |
| EVENT_FILTER_EVENT_ID  |
| Defines event IDs used in an EVENT_FILTER_DESCRIPTOR structure for an event ID or stack walk filter. |

## [EVENT\\_FILTER\\_EVENT\\_NAME](#)

Defines event IDs used in an EVENT\_FILTER\_DESCRIPTOR structure for an event name or stalk walk name filter.

## [EVENT\\_FILTER\\_HEADER](#)

Defines the header data that must precede the filter data that is defined in the instrumentation manifest.

## [EVENT\\_FILTER\\_LEVEL\\_KW](#)

Defines event IDs used in an EVENT\_FILTER\_DESCRIPTOR structure for a stack walk level-keyword filter.

## [EVENT\\_HEADER](#)

The EVENT\_HEADER structure (evntcons.h) defines information about the event.

## [EVENT\\_HEADER](#)

The EVENT\_HEADER structure (relogger.h) defines information about the event.

## [EVENT\\_HEADER\\_EXTENDED\\_DATA\\_ITEM](#)

The EVENT\_HEADER\_EXTENDED\_DATA\_ITEM structure (evntcons.h) defines the extended data that ETW collects as part of the event data.

## [EVENT\\_HEADER\\_EXTENDED\\_DATA\\_ITEM](#)

The EVENT\_HEADER\_EXTENDED\_DATA\_ITEM structure (relogger.h) defines the extended data that ETW collects as part of the event data.

## [EVENT\\_INSTANCE\\_HEADER](#)

The EVENT\_INSTANCE\_HEADER structure contains standard event tracing information common to all events written by TraceEventInstance.

## [EVENT\\_INSTANCE\\_INFO](#)

The EVENT\_INSTANCE\_INFO structure maps a unique transaction identifier to a registered event trace class for TraceEventInstance.

## [EVENT\\_MAP\\_ENTRY](#)

Defines a single value map entry.

## [EVENT\\_MAP\\_INFO](#)

Defines the metadata about the event map.

## [EVENT\\_PROPERTY\\_INFO](#)

Provides information about a single property of the event or filter.

## [EVENT\\_RECORD](#)

The EVENT\_RECORD structure (`evntcons.h`) defines the layout of an event that ETW delivers.

## [EVENT\\_RECORD](#)

The EVENT\_RECORD structure (`relogger.h`) defines the layout of an event that ETW delivers.

## [EVENT\\_TRACE](#)

The EVENT\_TRACE structure is used to deliver event information to an event trace consumer.

## [EVENT\\_TRACE\\_HEADER](#)

The EVENT\_TRACE\_HEADER structure contains standard event tracing information common to all events written by `TraceEvent`.

## [EVENT\\_TRACE\\_LOGFILEA](#)

The EVENT\_TRACE\_LOGFILEA (ANSI) structure (`evntrace.h`) stores information about a trace data source.

## [EVENT\\_TRACE\\_LOGFILEW](#)

The EVENT\_TRACE\_LOGFILEW (Unicode) structure (`evntrace.h`) stores information about a trace data source.

## [EVENT\\_TRACE\\_PROPERTIES](#)

The EVENT\_TRACE\_PROPERTIES structure contains information about an event tracing session and is used with APIs such as `StartTrace` and `ControlTrace`.

## [EVENT\\_TRACE\\_PROPERTIES\\_V2](#)

The EVENT\_TRACE\_PROPERTIES\_V2 structure contains information about an event tracing session and is used with APIs such as `StartTrace` and `ControlTrace`.

## [MOF\\_FIELD](#)

You may use the MOF\_FIELD structures to append event data to the EVENT\_TRACE\_HEADER or EVENT\_INSTANCE\_HEADER structures.

|   |  |
|---|--|
| <a href="#">PAYLOAD_FILTER_PREDICATE</a>  | Defines an event payload filter predicate that describes how to filter on a single field in a trace session.         |
| <a href="#">PROPERTY_DATA_DESCRIPTOR</a>  | Defines the property to retrieve.  |
| <a href="#">PROVIDER_ENUMERATION_INFO</a> | Defines the array of providers that have registered a MOF or manifest on the computer.                               |
| <a href="#">PROVIDER_EVENT_INFO</a>       | Defines an array of events in a provider manifest.   |
| <a href="#">PROVIDER_FIELD_INFO</a>       | Defines the field information.   |
| <a href="#">PROVIDER_FIELD_INFOARRAY</a>  | Defines metadata information about the requested field.  |
| <a href="#">PROVIDER_FILTER_INFO</a>      | Defines a filter and its data.   |
| <a href="#">TDH_CONTEXT</a>               | Defines the additional information required to parse an event.   |
| <a href="#">TRACE_ENABLE_INFO</a>         | Defines the session and the information that the session used to enable the provider.                                |
| <a href="#">TRACE_EVENT_INFO</a>          | Defines the information about the event.   |
| <a href="#">TRACE_GUID_INFO</a>           | Returned by <code>EnumerateTraceGuidsEx</code> . Defines the header to the list of sessions that enabled a provider. |
| <a href="#">TRACE_GUID_PROPERTIES</a>     | Returned by <code>EnumerateTraceGuids</code> . Contains information about an event trace provider.                   |

## [TRACE\\_GUID\\_REGISTRATION](#)

Used with RegisterTraceGuids to register event trace classes.

## [TRACE\\_LOGFILE\\_HEADER](#)

The TRACE\_LOGFILE\_HEADER structure contains information about an event tracing session and its events.

## [TRACE\\_PERIODIC\\_CAPTURE\\_STATE\\_INFO](#)

Used with TraceQueryInformation and TraceSetInformation to get or set information relating to a periodic capture state.

## [TRACE\\_PROVIDER\\_INFO](#)

Defines the GUID and name for a provider.

## [TRACE\\_PROVIDER\\_INSTANCE\\_INFO](#)

Defines an instance of the provider GUID.

## [TRACE\\_VERSION\\_INFO](#)

Determines the version information of the TraceLogging session.

# WNODE\_HEADER structure

Article • 08/19/2021 • 6 minutes to read

The **WNODE\_HEADER** structure is a member of the [EVENT\\_TRACE\\_PROPERTIES](#) structure.

## Syntax

C++

```
typedef struct _WNODE_HEADER {
    ULONG BufferSize;
    ULONG ProviderId;
    union {
        ULONG64 HistoricalContext;
        struct {
            ULONG Version;
            ULONG Linkage;
        };
    };
    union {
        HANDLE KernelHandle;
        LARGE_INTEGER TimeStamp;
    };
    GUID Guid;
    ULONG ClientContext;
    ULONG Flags;
} WNODE_HEADER, *PWNODE_HEADER;
```

## Members

### BufferSize

Total size of memory allocated, in bytes, for the event tracing session properties. The size of memory must include the room for the [EVENT\\_TRACE\\_PROPERTIES](#) structure plus the session name string and log file name string that follow the structure in memory.

### ProviderId

Reserved for internal use.

### HistoricalContext

On output, the handle to the event tracing session.

## **Version**

Reserved for internal use.

## **Linkage**

Reserved for internal use.

## **KernelHandle**

Reserved for internal use.

## **TimeStamp**

Time at which the information in this structure was updated, in 100-nanosecond intervals since midnight, January 1, 1601.

## **Guid**

The GUID that you define for the session.

For an NT Kernel Logger session, set this member to **SystemTraceControlGuid**.

If this member is set to **SystemTraceControlGuid** or **GlobalLoggerGuid**, the logger will be a system logger.

For a private logger session, set this member to the provider's GUID that you are going to enable for the session.

If you start a session that is not a kernel logger or private logger session, you do not have to specify a session GUID. If you do not specify a GUID, ETW creates one for you. You need to specify a session GUID only if you want to change the default permissions associated with a specific session. For details, see the EventAccessControl function.

You cannot start more than one session with the same session GUID.

**Prior to Windows Vista:** You can start more than one session with the same session GUID.

## **ClientContext**

Clock resolution to use when logging the time stamp for each event. The default is Query performance counter (QPC).

**Prior to Windows Vista:** The default is system time.

**Prior to Windows 10, version 1703:** No more than 2 distinct clock types can be used simultaneously by any system loggers.

**Starting with Windows 10, version 1703:** The clock type restriction has been removed.

All three clock types can now be used simultaneously by system loggers.

You can specify one of the following values.

| Value | Meaning  |
|-------|--|
| 1     | <p>Query performance counter (QPC). The QPC counter provides a high-resolution time stamp that is not affected by adjustments to the system clock. The time stamp stored in the event is equivalent to the value returned from the QueryPerformanceCounter API. For more information on the characteristics of this time stamp, see <a href="#">Acquiring high-resolution time stamps</a>.</p> <p>You should use this resolution if you have high event rates or if the consumer merges events from different buffers. In these cases, the precision and stability of the QPC time stamp allows for better accuracy in ordering the events from different buffers. However, the QPC time stamp will not reflect updates to the system clock, e.g. if the system clock is adjusted forward due to synchronization with an NTP server while the trace is in progress, the QPC time stamps in the trace will continue to reflect time as if no update had occurred.</p> <p>To determine the resolution, use the <b>PerfFreq</b> member of <a href="#">TRACE_LOGFILE_HEADER</a> when consuming the event.</p> <p>To convert an event's time stamp into 100-ns units, use the following conversion formula:<br/>scaledTimestamp = eventRecord.EventHeader.TimeStamp.QuadPart * 10000000.0 / logfileHeader.PerfFreq.QuadPart</p> <p>Note that on older computers, the time stamp may not be accurate because the counter sometimes skips forward due to hardware errors.</p> |

| Value | Meaning   |
|-------|---|
| 2     | <p>System time. The system time provides a time stamp that tracks changes to the system's clock, e.g. if the system clock is adjusted forward due to synchronization with an NTP server while the trace is in progress, the System Time time stamps in the trace will also jump forward to match the new setting of the system clock.</p> <ul style="list-style-type: none"> <li>• On systems prior to Windows 10, the time stamp stored in the event is equivalent to the value returned from the <code>GetSystemTimeAsFileTime</code> API.</li> <li>• On Windows 10 or later, the time stamp stored in the event is equivalent to the value returned from the <code>GetSystemTimePreciseAsFileTime</code> API.</li> </ul> <p>Prior to Windows 10, the resolution of this time stamp was the resolution of a system clock tick, as indicated by the <code>TimerResolution</code> member of <code>TRACE_LOGFILE_HEADER</code>. Starting with Windows 10, the resolution of this time stamp is the performance counter resolution, as indicated by the <code>PerfFreq</code> member of <code>TRACE_LOGFILE_HEADER</code>. To convert an event's time stamp into 100-ns units, use the following conversion formula:</p> $\text{scaledTimestamp} = \text{eventRecord.EventHeader.TimeStamp.QuadPart}$ <p>Note that when events are captured on a system running an OS prior to Windows 10, if the volume of events is high, the resolution for system time may not be fine enough to determine the sequence of events. In this case, a set of events will have the same time stamp, but the order in which ETW delivers the events may not be correct. Starting with Windows 10, the time stamp is captured with additional precision, though some instability may still occur in cases where the system clock was adjusted while the trace was being captured.</p> |
| 3     | <p>CPU cycle counter. The CPU counter provides the highest resolution time stamp and is the least resource-intensive to retrieve. However, the CPU counter is unreliable and should not be used in production. For example, on some computers, the timers will change frequency due to thermal and power changes, in addition to stopping in some states.</p> <p>To determine the resolution, use the <code>CpuSpeedInMHz</code> member of <code>TRACE_LOGFILE_HEADER</code> when consuming the event.</p> <p>If your hardware does not support this clock type, ETW uses system time.</p> <p><b>Windows Server 2003, Windows XP with SP1 and Windows XP:</b> This value is not supported, it was introduced in Windows Server 2003 with SP1 and Windows XP with SP2.</p>   |

**Windows 2000:** The `ClientContext` member is not supported.

## Flags

Must contain `WNODE_FLAG_TRACED_GUID` to indicate that the structure contains event tracing information.

## Remarks

Be sure to initialize the memory for this structure to zero before setting any members.

To convert an ETW timestamp into a FILETIME, use the following procedure:

1. For each session or log file being processed (i.e. for each EVENT\TRACE\LOGFILE), check the `logFile.ProcessTraceMode` field to determine whether the `PROCESS\TRACE\MODE\RAW\TIMESTAMP` flag is set. By default, this flag is not set. If this flag is not set, the ETW runtime will automatically convert the timestamp of each EVENT\RECORD into a FILETIME before sending the EVENT\RECORD to your `EventRecordCallback` function, so no additional processing is needed. The following steps should only be used if the trace is being processed with the `PROCESS\TRACE\MODE\RAW\TIMESTAMP` flag set.

2. For each session or log file being processed (i.e. for each EVENT\TRACE\LOGFILE), check the `logFile.LogfileHeader.ReservedFlags` field to determine the time stamp scale for the log file. Based on the value of `ReservedFlags`, follow one of these steps to determine the value to use for `timeStampScale` in the remaining steps:

- a. If `ReservedFlags == 1` (QPC): `DOUBLE timeStampScale = 10000000.0 / logFile.LogfileHeader.PerfFreq.QuadPart;`
- b. If `ReservedFlags == 2` (System time): `DOUBLE timeStampScale = 1.0;` Note that the remaining steps are unnecessary for events using system time, since the events already provide their time stamps in FILETIME units. The remaining steps will work but are unnecessary and will introduce a small rounding error.
- c. If `ReservedFlags == 3` (CPU cycle counter): `DOUBLE timeStampScale = 10.0 / logFile.LogfileHeader.CpuSpeedInMHz;`

3. On the FIRST call to your `EventRecordCallback` function for a particular log file, use data from the `logFile` (EVENT\TRACE\LOGFILE) and from the `eventRecord` (EVENT\RECORD) to compute the `timeStampBase` that will be used for the remaining events in the log file: `INT64 timeStampBase = logFile.LogfileHeader.StartTime.QuadPart - (INT64)(timeStampScale * eventRecord.EventHeader.TimeStamp.QuadPart);`

4. For each `eventRecord` (EVENT\RECORD), convert the event's timestamp into FILETIME as follows, using the `timeStampScale` and `timeStampBase` values calculated in steps 2 and 3: `INT64 timeStampInFileTime = timeStampBase + (INT64)(timeStampScale * eventRecord.EventHeader.TimeStamp.QuadPart);`

## Requirements

| Requirement              | Value   |
|--------------------------|---|
| Minimum supported client | Windows 2000 Professional [desktop apps   UWP apps] |
| Minimum supported server | Windows 2000 Server [desktop apps   UWP apps]       |
| Header                   | Wmistr.h  |

## See also

*ControlCallback*

[EVENT\\_TRACE\\_PROPERTIES](#)

[GetTraceLoggerHandle](#)

[LARGE\\_INTEGER](#)