

# Event Logging

Many applications record errors and events in various proprietary error logs. These proprietary error logs have different formats and display different user interfaces. Moreover, you cannot merge the data to provide a complete report. Therefore, you need to check a variety of sources to diagnose problems. Event logging in Microsoft® Windows NT® provides a standard, centralized way for applications (and the operating system) to record important software and hardware events. The event-logging service stores events from various sources in a single collection called an *event log*. Windows NT also supplies Event Viewer for viewing the logs, and a programming interface for examining the logs.

This overview contains the following topics, which discuss the programming interface for writing to and examining event logs.

- [Event types](#)
- [Logging guidelines](#)
- [Event logging elements](#)
- [Event logging operations](#)
- [Event logging model](#)
- [Event logging security](#)

## About Event Logging

When an error occurs, the system administrator or support technicians must determine what caused the error, attempt to recover any lost data, and prevent the error from recurring. It is helpful if applications, the operating system, and other system services record important events such as low-memory conditions or excessive attempts to access a disk. Then the system administrator can use the event log to help determine what conditions caused the error and the context in which it occurred. By periodically viewing the event log, the system administrator may be able to identify problems (such as a failing hard drive) before they cause damage.

## Event Types

Windows NT defines five types of events that can be logged. All event classifications have well-defined common data and can optionally include event-specific data. The application indicates the event type when it reports an event. Each event must be of a single type. The Event Viewer uses this type to determine which icon to display in the list view of the log.

The following table describes the event types used in event logging.

<b>Event type</b>	<b>Description</b>
Information	Information events indicate infrequent but significant successful operations. For example, when Microsoft® SQL Server™ successfully loads, it may be appropriate to log an information event stating that "SQL Server has started." Note that while this is appropriate behavior for major server services, it is generally inappropriate for a desktop application (Microsoft® Excel, for example) to log an event each time it starts.
Warning	Warning events indicate problems that are not immediately significant, but that may indicate conditions that could cause future problems. Resource consumption is a good candidate for a warning event. For example, an application can log a warning event if disk space is low. If an application can recover from an event without loss of functionality or data, it can generally classify the event as a warning event.
Error	Error events indicate significant problems that the user should know about. Error events usually indicate a loss of functionality or data. For example, if a service cannot be loaded as the system boots, it can log an error event.
Success audit	Success audit events are security events that occur when an audited access attempt is successful. For example, a successful logon attempt is a success audit event.
Failure audit	Failure audit events are security events that occur when an audited access attempt fails. For example, a failed attempt to open a file is a failure audit event.

Selected activities of users on can be tracked by auditing security events and then placing entries in a computer's security log. To determine the types of security events that will be logged for the computer, use the Windows NT User Manager. For more information, look up "auditing" in the online help provided with the Windows NT User Manager.

## Logging Guidelines

Event logs store records of significant events on behalf of Windows NT and applications running on Windows NT. Because the logging functions are general purpose, you must decide what information is appropriate to log. Generally, you should log only information that could be useful in diagnosing a hardware or software problem. Event logging is not intended to be used as a tracing tool.

The following are examples of cases in which event logging can be helpful.

- Resource problems. If an application gets into a low-memory situation (caused by a code bug or inadequate memory) that degrades performance, logging a warning event when memory allocation fails might provide a clue about what went wrong.
- Hardware problems. If a device driver encounters a disk controller time-out, a power failure in a parallel port, or a data error from a network or serial card, logging information about these events can help the system administrator diagnose hardware problems. The device driver logs the error.
- Bad sectors. If a disk driver encounters a bad sector, it may be able to read from or write to the sector after retrying the operation, but the sector will go bad eventually. Therefore, if the

disk driver can proceed, it should log a warning; otherwise, it should log an error event. If a file system driver finds a large number of bad sectors, fixes them, and logs warning events, logging information of this type might indicate that the disk is about to fail.

- Information events. A server application (such as a database server) records a user logging on, opening a database, or starting a file transfer. The server can also log error events it encounters (cannot access file, host process disconnected, and so on), a corruption in the database, or whether a file transfer was successful.

Event logging consumes resources such as disk space and processor time. The amount of disk space that an event log requires and the overhead for an application that logs events depend on how much information you choose to log. This is why it is important to log only essential information. It is also good to place event logging calls in an error path in the code rather than in the main code path, which would reduce performance.

The amount of disk space required per event log record includes the members of the **EVENTLOGRECORD** structure. This is a variable length structure; strings and binary data are stored following the structure.

## Event Logging Elements

The following are the major elements used in event logging:

- [Logfiles](#)
- [Event sources](#)
- [Event categories](#)
- [Event identifiers](#)
- [Message files](#)
- [Event data](#)

### Logfiles

The event-logging service uses the information stored in the **EventLog** registry key. The **EventLog** key (shown in the following example) contains several subkeys, called *logfiles*. Logfile registry information is used to locate resources that the event logging service needs when an application writes to and reads from the event log. The default logfiles are **Application**, **Security**, and **System**. The structure is as follows:

```
HKEY_LOCAL_MACHINE
SYSTEM
  CurrentControlSet
    Services
      EventLog
        Application
        Security
        System
```

Applications and services use the **Application** logfile. Device drivers use the **System** logfile. Windows NT will generate success and failure audit events in the **Security** log when auditing is

turned on. For more information about auditing security events, see the documentation for the Windows NT User Manager.

## Event Sources

Each [logfile](#) can contain subkeys called *event sources*. The event source is the name of the software that logs the event. It is often the name of the application, or the name of a subcomponent of the application, if the application is large. Applications and services should add their names to the **Application** logfile. Device drivers should add their names to the **System** logfile. The structure is as follows:

```
HKEY_LOCAL_MACHINE
  SYSTEM
    CurrentControlSet
      Services
        EventLog
          Application
            AppName
          Security
          System
            DriverName
```

The application provides its name when it opens the event log using the [RegisterEventSource](#) function. You cannot use a source name that has already been used as a logfile name. In addition, source names cannot be hierarchical (that is, you cannot use the backslash character [\\]).

Each event source contains information specific to the software that will be logging the events, such as the [message files](#), as shown in the following table.

Registry Value	Description
<b>EventMessageFile</b>	Specifies the path for the event message file. You can list multiple files, separated by semicolons. An event message file contains language-dependent strings that describe the events. This value has the type REG_EXPAND_SZ.
<b>CategoryMessageFile</b>	Specifies the path for the category message file. You can list multiple files, separated by semicolons. A category message file contains language-dependent strings that describe the categories. This value has the type REG_EXPAND_SZ.
<b>ParameterMessageFile</b>	Specifies the path for the parameter message file. You can list multiple files, separated by semicolons. A parameter message file contains language-independent strings that are to be inserted into the event description strings. This value has the type REG_EXPAND_SZ.
<b>CategoryCount</b>	Specifies the number of event categories supported. This value has the type REG_DWORD.

**TypesSupported**

Specifies a bitmask of supported types. This value has the type REG\_DWORD. It can be one or more of the following values:

EVENTLOG\_ERROR\_TYPE  
EVENTLOG\_WARNING\_TYPE  
EVENTLOG\_INFORMATION\_TYPE  
EVENTLOG\_AUDIT\_SUCCESS  
EVENTLOG\_AUDIT\_FAILURE

When an application uses the [RegisterEventSource](#) or [OpenEventLog](#) function to get a handle to an event log, the event-logging service searches for the specified event source in the registry. For example, the **Application** logfile might contain event sources for Microsoft SQL Server and Microsoft Excel. If an application uses **RegisterEventSource** or **OpenEventLog** with a source name of Application, SQL, or Excel, the event-logging service returns a handle to the **Application** logfile.

An application can use the **Application** event log without adding a new event source to the registry. If the application calls **RegisterEventSource**, passing a source name that cannot be found in the registry, the event-logging service uses the **Application** logfile by default. However, because there are no message files, the Event Viewer cannot map any event identifiers or event categories to a description string, and will display an error. For this reason, you should add a unique event source to the registry for your application and specify a message file.

**Event Categories**

Categories help you organize events so Event Viewer can filter them. Each [event source](#) can define its own numbered categories and the text strings to which they are mapped. The categories must be numbered consecutively beginning with the number 1. For example, the security system uses the following categories:

- Logon/logoff
- File system access
- Privileged actions
- Change in security policy

**Event Identifiers**

*Event identifiers* uniquely identify a particular event. Each [event source](#) can define its own numbered events and the description strings to which they are mapped. Event viewers can present these strings to the user. They should help the user understand what went wrong and suggest what actions to take. Direct the description at users solving their own problems, not at administrators or support technicians. Make the description clear and concise and avoid culture-specific phrases.

The following sections discuss these description strings, and the insertion strings that serve as placeholders in the description strings.

**Description Strings**

The description strings in the event message file are indexed by event identifier, enabling Event Viewer to display event-specific text for any event based on the event identifier. All descriptions are localized and language dependent. The description strings may contain insertion string placeholders, of the form %*n*, where %1 indicates the first insertion string, and so on. For example, the following is a sample entry in the .MC file:

```
MessageId=0x4
Severity=Error
Facility=System
SymbolicName=MSG_CMD_DELETE
Language=English
File %1 contains %2, which is in error.
```

In this case, the buffer returned by [ReadEventLog](#) contains insertion strings. The **NumStrings** member of the **EVENTLOGRECORD** structure indicates the number of insertion strings. The **StringOffset** member of the **EVENTLOGRECORD** structure indicates the location of the first insertion string in the buffer.

The description string can also contain placeholders for parameter strings from the parameter message file. The placeholders are of the form %%*n*, where %%1 is replaced by the parameter string with the identifier of 1, and so on. In this case, the event viewer uses [LoadLibraryEx](#) and [FormatMessage](#) to retrieve the insertion string from the file indicated by the source's **ParameterMessageFile** registry value.

## Insertion Strings

Insertion strings are optional language-independent strings used to fill in values for placeholders in description strings. Because the strings are not localized, it is critical that these placeholders be used only to represent language-independent strings such as numeric values, file names, user names, and so on. The string length must not exceed 32 kilobytes – 1 characters.

It is not acceptable to use several strings to create a larger description. The insertion string should be treated as data, not text. For example, the following example is *not* recommended:

```
LPSTR pszString1 = "successfully";
LPSTR pszString2 = "not";
LPSTR pszDescription = "The user was %1 added to the database.";
```

It is not acceptable to use `pszString1` and `pszString2` to form the strings "The user was successfully added to the database." and "The user was not added to the database." There are three reasons this substitution is not effective:

- The strings "successfully" and "not" should be localized.
- Even if the description strings are obtained from language-dependent message files, this is done when the event is logged, not when it is viewed. When the event is viewed, the language may be wrong.
- Such substitutions of adverbs and adjectives will not work in many other languages. The preceding example should use two separate events, each with its own description string.

In the following example, it is appropriate to use either `pszString1` or `pszString2` for the insertion string in `pszDescription`.

```
LPSTR pszString1 = "c:\\testappl.c";
LPSTR pszString2 = "c:\\testapp2.c";
LPSTR pszDescription = "Access denied. Attempted to open the file %1."
```

## Message Files

Each [event sources](#) should register *message files* that contain description strings for each [event identifier](#), [event category](#), and [parameter](#). Register these files in the **EventMessageFile**, **CategoryMessageFile**, and **ParameterMessageFile** registry values for the event source. You can create one message file that contains descriptions for the event identifiers, categories, and parameters, or create three separate message files. Several applications can share the same message file.

You should typically create message files as dynamic-link libraries (DLL). Use the following procedure to create these DLLs.

### ▶ To create a message file

1. Do not include exported functions in the .C file; include only a stub for the [DllMain](#) function. The stub should simply return TRUE.
2. Use the following command to compile the .C file:  
**cl options -fo filename.obj filename.c.**
3. Create an .MC file to define the message resource table. For more information, see [Message Compiler Source Files](#).
4. Use the message compiler to create .RC and .BIN files from the .MC file. Use the following command: **mc filename.mc.**
5. Use the resource compiler to create a .RES file. Use the following command: **rc -r -fo filename.res filename.rc.**
6. Use the linker to create a .DLL file. Use the following command line: **link -dll -out:filename.dll filename.obj filename.res.**

To make it easier for the application to use your message file, create a header file that lists each event. For example, suppose you have defined the following message in your .MC file:

```
MessageId=0x4
Severity=Error
Facility=System
SymbolicName=MSG_CMD_DELETE
Language=English
File %1 contains %2, which is in error.
```

Your header file should contain the following code:

```
//
// MessageId: MSG_CMD_DELETE
// MessageText:
// File %1 contains %2, which is in error.
//
#define MSG_CMD_DELETE ((DWORD)0xC000004L)
```

Now the event-viewing application can use the following procedure to gain access to the [description strings](#) in the message file.

► **To obtain description strings**

1. Use the [RegOpenKey](#) function to open the event source.
2. Use the [RegQueryValueEx](#) function to obtain the **EventMessageFile** value for the event source, which is the name of the event message DLL.
3. Use the [LoadLibraryEx](#) function to load the event message DLL.
4. Use the [FormatMessage](#) function to obtain the description from the DLL and add the insertion strings.

## Event Data

Each event can have event-specific data associated with it. The Event Viewer does not interpret this data; it displays extra data only in a combined hexadecimal and text format. Use event-specific data sparingly, including it only if you are sure it will be useful to someone debugging the problem. For example, many network-related events include network control blocks (NCBs) as event-specific data.

When you use event-specific data, the last part of its description string should provide a note about the information provided as event-specific data. For example, the network software could provide a note such as: "(The NCB is the event data)." As a convention, use parentheses around such remarks, as indicated in this example.

You can also use event-specific data to store information the application can process independently of the Event Viewer. For example, you could write a viewer specifically for your events, or write a program that scans the logfile and makes reports that include information from the event-specific data.

## Event Logging Operations

The [OpenEventLog](#), [OpenBackupEventLog](#), [RegisterEventSource](#), [DeregisterEventSource](#), and [CloseEventLog](#) functions open and close event log handles. Accessing the object through its handle provides an object-oriented model for the event-logging functions, as well as some performance gain when requesting multiple operations on the logs.

The following table shows the operations that can be performed on an open event log, and the corresponding function for each operation.

<b>Operation</b>	<b>Function</b>
Backup	<a href="#">BackupEventLog</a>
Clear	<a href="#">ClearEventLog</a>
Monitor	<a href="#">NotifyChangeEventLog</a>
Query	<a href="#">GetOldestEventLogRecord</a> , <a href="#">GetNumberOfEventLogRecords</a>

Read	<a href="#">ReadEventLog</a>
Write	<a href="#">ReportEvent</a>

The [OpenEventLog](#) and [ReportEvent](#) functions take an optional server name as a parameter so the operations can be performed on the remote server. Use **OpenEventLog** for reading or performing administrative operations (backup, clear, monitor, and query) on the log, and use [RegisterEventSource](#) for writing to the log.

Each call to an event logging function is an atomic operation. When you read from the event log, only whole event records are returned. When you write to the event log, each event record is guaranteed to be written sequentially as a complete record in the log. The following list describes how the event-logging service handles special conditions:

- The event-logging service receives a read operation and a write operation at the same time. If the read position is at the end of the file, either the read operation fails with an "end-of-file" status (if the write operation has not been completed), or it returns the new record (if the write operation has been completed).
- The event-logging service completes a clear operation before receiving a read operation. The read operation fails with "end-of-file" status.
- The event-logging service completes a clear operation before receiving a write operation. The clear operation truncates the log, then the write operation adds the new record at the beginning of the log.

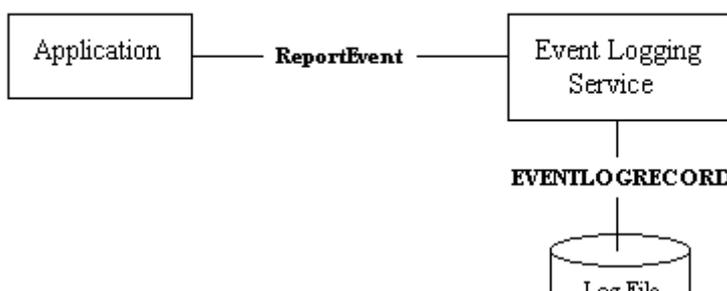
## Event Logging Model

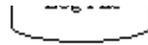
The following sections describe the processes that form the event-logging model.

- [Writing to the Event Log](#)
- [Reading from the Event Log](#)
- [Viewing the Event Log](#)

### Writing to the Event Log

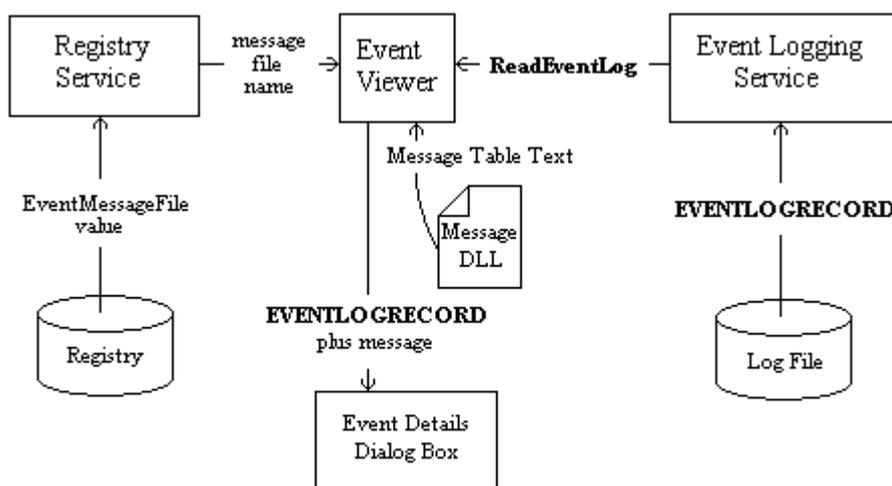
When an application calls the [ReportEvent](#) function to write an entry to the log, Windows NT passes the parameters to the event-logging service. The event-logging service uses the information to write an [EVENTLOGRECORD](#) structure to the event log. The following diagram illustrates this process.





## Reading from the Event Log

An event viewer application uses the [OpenEventLog](#) function to open the event log for an event source. The event viewer can then use the [ReadEventLog](#) function to read event records from the log. [ReadEventLog](#) returns a buffer containing an [EVENTLOGRECORD](#) structure and additional information that describes a logged event. The following diagram illustrates this process.



## Viewing the Event Log

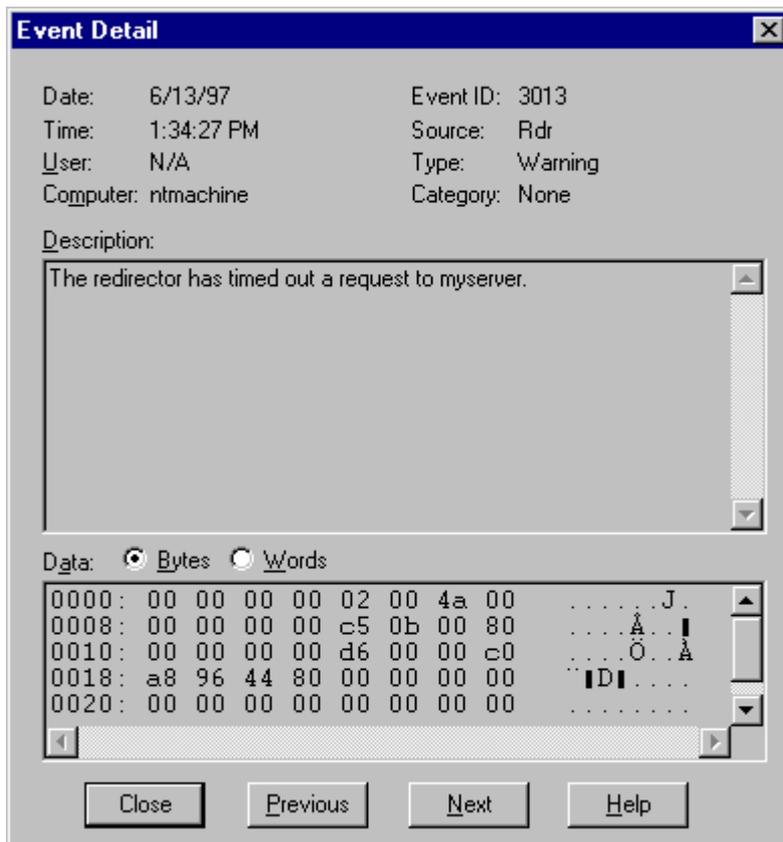
When the user starts Event Viewer to view the event log entries, it calls the [ReadEventLog](#) function to obtain the [EVENTLOGRECORD](#) structures. The Event Viewer uses the event source and event identifier to get message text for each event from the registered message file (indicated by the [EventMessageFile](#) registry value for the source). The Event Viewer uses the [LoadLibraryEx](#) function to load the message file. The Event Viewer then uses the [FormatMessage](#) function to retrieve the description string from the loaded module.

The following illustration shows how the Event Viewer presents this information.

Date	Time	Source	Category	Event	User	Computer
5/2/97	1:50:56 PM	EventLog	None	6005	N/A	ntrmachine
5/2/97	1:50:58 PM	EI90x	None	0	N/A	ntrmachine
5/1/97	5:21:03 PM	Rdr	None	3009	N/A	ntrmachine
5/1/97	5:21:03 PM	Rdr	None	3009	N/A	ntrmachine
5/1/97	5:21:03 PM	Rdr	None	3009	N/A	ntrmachine

!	5/1/97	5:21:03 PM	Rdr	None	3009	N/A	ntmachine
!	5/1/97	5:21:02 PM	Rdr	None	3009	N/A	ntmachine
!	5/1/97	5:21:02 PM	Rdr	None	3009	N/A	ntmachine
!	5/1/97	5:21:02 PM	Rdr	None	3009	N/A	ntmachine
!	4/30/97	3:08:10 PM	Print	None	20	ntuser	ntmachine
i	4/29/97	11:35:49 AM	EI90x	None	3	N/A	ntmachine
i	4/29/97	11:35:49 AM	EI90x	None	3	N/A	ntmachine
i	4/29/97	11:35:49 AM	EI90x	None	3	N/A	ntmachine
i	4/29/97	11:35:46 AM	EventLog	None	6005	N/A	ntmachine
i	4/29/97	11:35:48 AM	EI90x	None	0	N/A	ntmachine
i	4/29/97	9:37:36 AM	EI90x	None	3	N/A	ntmachine
i	4/29/97	9:37:36 AM	EI90x	None	3	N/A	ntmachine
i	4/29/97	9:37:36 AM	EI90x	None	3	N/A	ntmachine
i	4/29/97	9:37:34 AM	EventLog	None	6005	N/A	ntmachine
i	4/29/97	9:37:36 AM	EI90x	None	0	N/A	ntmachine
!	4/28/97	9:21:59 PM	NETLOGON	None	5719	N/A	ntmachine
i	4/28/97	9:21:29 PM	EI90x	None	3	N/A	ntmachine
i	4/28/97	9:21:29 PM	EI90x	None	3	N/A	ntmachine
i	4/28/97	9:21:29 PM	EI90x	None	3	N/A	ntmachine

If the user double-clicks on an event log entry, the Event Viewer displays more information, as shown in the following illustration.



## Event Logging Security

Access to the event logs is determined by the account under which the application is running. The LocalSystem account is a special account that Windows NT services can use. The Administrator account consists of the administrators for the system. The Server Operator account (ServerOp) consists of the administrators of the domain server. The World account includes all users on all

systems.

The following table shows which accounts are granted read, write, and clear access to each log.

<b>Log</b>	<b>Account</b>	<b>Access</b>		
<b>Application</b>	LocalSystem	Read	Write	Clear
	Adminstrator	Read	Write	Clear
	ServerOp	Read	Write	Clear
	World	Read	Write	
<b>Security</b>	LocalSystem	Read	Write	Clear
	Adminstrator	Read		Clear
	World	Read		Clear
<b>System</b>	LocalSystem	Read	Write	Clear
	Adminstrator	Read	Write	Clear
	ServerOp	Read		Clear
	World	Read		

In addition, users can read and clear the **Security** log if they have been granted one of the following:

- The "manage auditing and security log" user right. Use the Windows NT User Manager. Click the **Policies** menu, then click **User Rights**.
- The SE\_AUDIT\_NAME privilege. For more information, see [Windows NT Privileges](#).

The following table shows which types of access are required for each event logging function:

<b>Function</b>	<b>Access Required</b>
<b>OpenEventLog</b>	Read
<b>OpenBackupEventLog</b>	Read
<b>RegisterEventSource</b>	Write
<b>ClearEventLog</b>	Clear

As an example, **OpenEventLog** requires read access. A member of the ServerOp account can call **OpenEventLog** for the **Application** event log and the **System** event log, because ServerOp has read access for both of these logs. However, a member of the ServerOp account cannot call **OpenEventLog** for the **Security** log, because it does not have read access for this log.

## Using Event Logging

You can use the Windows NT Registry Editor (REGEDT32.EXE) to view the event log registry keys, and you can use the Windows NT Event Viewer to view the event logs. To create event log entries, use the event logging functions as shown in the following topics.

- [Adding a source to the registry](#)

- [Reporting an event](#)
- [Querying the event log](#)
- [Reading the event log](#)
- [Displaying the user for an event](#)
- [Displaying the local time for an event](#)

## Adding a Source to the Registry

You can use the default **Application** event log without adding an event source to the registry. However, Event Viewer will not be able to map your event identifier codes to message strings unless you register your event source and provide a message file.

You can add a new source name to the registry by opening a new registry subkey under the **Application** key using the [RegCreateKey](#) function, and adding registry values to the new subkey using the [RegSetValueEx](#) function. The following example opens a new source name called `SamplApp` and adds a message-file name and a bitmask of supported types.

```
void AddEventSource()
{
    HKEY hk;
    DWORD dwData;
    UCHAR szBuf[80];

    // Add your source name as a subkey under the Application
    // key in the EventLog registry key.

    if (RegCreateKey(HKEY_LOCAL_MACHINE,
        "SYSTEM\\CurrentControlSet\\Services\\
        \\EventLog\\Application\\SamplApp", &hk))
        ErrorExit("Could not create the registry key.");

    // Set the name of the message file.

    strcpy(szBuf, "%SystemRoot%\\System\\SamplApp.dll");

    // Add the name to the EventMessageFile subkey.

    if (RegSetValueEx(hk,          // subkey handle
        "EventMessageFile",      // value name
        0,                        // must be zero
        REG_EXPAND_SZ,          // value type
        (LPBYTE) szBuf,          // pointer to value data
        strlen(szBuf) + 1))      // length of value data
        ErrorExit("Could not set the event message file.");

    // Set the supported event types in the TypesSupported subkey.

    dwData = EVENTLOG_ERROR_TYPE | EVENTLOG_WARNING_TYPE |
        EVENTLOG_INFORMATION_TYPE;

    if (RegSetValueEx(hk,          // subkey handle
        "TypesSupported",         // value name
        0,                        // must be zero
        REG_DWORD,                // value type
        (LPBYTE) &dwData,        // pointer to value data
        sizeof(DWORD))            // length of value data
        ErrorExit("Could not set the supported types.");

    RegCloseKey(hk);
}
```

## Reporting an Event

After you have added a source name to the registry, use the [RegisterEventSource](#) function to get a handle to the **Application** event log. The following example obtains the handle and then adds an event to the log using the [ReportEvent](#) function.

```
void MyReportEvent(LPSTR szMsg)
{
    HANDLE h;

    h = RegisterEventSource(NULL, // uses local computer
        "SamplApp");           // source name
    if (h == NULL)
        ErrorExit("Could not register the event source.");

    if (!ReportEvent(h,          // event log handle
        EVENTLOG_ERROR_TYPE,    // event type
        0,                       // category zero
        MSG_ERR_EXIST,          // event identifier
        NULL,                    // no user security identifier
        1,                       // one substitution string
        0,                       // no data
        (LPTSTR *) szMsg,       // pointer to string array
        NULL))                   // pointer to data
        ErrorExit("Could not report the event.");

    DeregisterEventSource(h);
}
```

Recall that your header file contains the event identifiers. For this example, the following event identifier was used:

```
//
// MessageId: MSG_ERR_EXIST
// MessageText:
// File %1 does not exist.
//
#define MSG_ERR_EXIST ((DWORD)0xC0000004L)
```

## Querying the Event Log

If you want to find out how many event records are in the event log, use the [GetNumberOfEventLogRecords](#) function. The following example displays the number of event records in the **Application** event log and the **System** event log. It opens the logs using the [OpenEventLog](#) function and obtains the number of records using [GetNumberOfEventLogRecords](#).

```
void DisplayEventCount()
{
    HANDLE h;
    DWORD cRecords;
```

```

// Open the System log.

h = OpenEventLog(NULL, // uses local computer
                "System"); // source name
if (h == NULL)
    ErrorExit("Could not open the System event log.");

// Get the number of records in the System event log.

if (!GetNumberOfEventLogRecords(h, &cRecords))
    ErrorExit("Could not get the number of records.");

printf("There are %d records in the System event log.\n",
       cRecords);

CloseEventLog(h);

// Open the Application log.

h = OpenEventLog(NULL, // uses local computer
                "Application"); // source name
if (h == NULL)
    ErrorExit("Could not open the Application event log.");

// Get the number of records in the Application event log.

if (!GetNumberOfEventLogRecords(h, &cRecords))
    ErrorExit("Could not get number of records.");

printf("There are %d records in the Application event log.\n",
       cRecords);

CloseEventLog(h);
}

```

## Reading the Event Log

The following example reads all the records in the **Application** log file and displays the event identifier, event type, and event source for each event log entry.

```

void DisplayEntries( )
{
    HANDLE h;
    EVENTLOGRECORD *pevlr;
    BYTE bBuffer[BUFFER_SIZE];
    DWORD dwRead, dwNeeded, cRecords, dwThisRecord = 0;

    // Open the Application event log.

    h = OpenEventLog( NULL, // use local computer
                    "Application"); // source name
    if (h == NULL)
        ErrorExit("Could not open the Application event log.");

    pevlr = (EVENTLOGRECORD *) &bBuffer;

    // Opening the event log positions the file pointer for this
    // handle at the beginning of the log. Read the records
    // sequentially until there are no more.

    while (ReadEventLog(h, // event log handle
                       EVENTLOG_FORWARDS_READ | // reads forward
                       EVENTLOG_SEQUENTIAL_READ, // sequential read

```

```

        0,           // ignored for sequential reads
        pevlr,      // pointer to buffer
        BUFFER_SIZE, // size of buffer
        &dwRead,    // number of bytes read
        &dwNeeded)) // bytes in next record
    {
        while (dwRead > 0)
        {
            // Print the event identifier, type, and source name.
            // The source name is just past the end of the
            // formal structure.

            printf("%02d Event ID: 0x%08X ",
                dwThisRecord++, pevlr->EventID);
            printf("EventType: %d Source: %s\n",
                pevlr->EventType, (LPSTR) ((LPBYTE) pevlr +
                sizeof(EVENTLOGRECORD)));

            dwRead -= pevlr->Length;
            pevlr = (EVENTLOGRECORD *)
                ((LPBYTE) pevlr + pevlr->Length);
        }

        pevlr = (EVENTLOGRECORD *) &bBuffer;
    }

    CloseEventLog(h);
}

```

## Displaying the User for an Event

The following example retrieves the name of the user for an event. The function parameters are a pointer to the [EVENTLOGRECORD](#) structure, a pointer to a buffer to receive the user name, and a pointer to the size of the allocated buffer. If the function succeeds, it returns TRUE; otherwise, it returns FALSE. To get extended error information, call [GetLastError](#).

```

BOOL
GetEventUserName(EVENTLOGRECORD *pelr, LPSTR pszUser, LPDWORD pcbUser)
{
    PSID lpSid;
    char szName[256];
    char szDomain[256];
    SID_NAME_USE snu;
    DWORD dwLen;
    DWORD cbName = 256;
    DWORD cbDomain = 256;

    // Point to the SID.
    lpSid = (PSID)((LPBYTE) pelr + pelr->UserSidOffset);

    if (LookupAccountSid(NULL, lpSid, szName, &cbName, szDomain,
        &cbDomain, &snu))
    {
        // Determine whether the buffer is large enough.
        dwLen = lstrlen( pszUser ) + 1;

        if (dwLen > *lpcbUser)
        {
            SetLastError( ERROR_INSUFFICIENT_BUFFER );
            *lpcbUser = dwLen;
            return FALSE;
        }
    }
}

```

```
        // Return the user's name.
        lstrcpy( lpszUser, szName );
    }
    else
    {
        // Use the error status from LookupAccountSid.
        return FALSE;
    }

    SetLastError(0);
    return TRUE;
}
```

## Displaying the Local Time for an Event

The following example displays the time information for an event. The function parameter is a pointer to the [EVENTLOGRECORD](#) structure. The function has no return value.

```
void PrintTimeGenerated(EVENTLOGRECORD *pevlr)
{
    FILETIME FileTime, LocalFileTime;
    SYSTEMTIME SysTime;
    __int64 lgTemp;
    __int64 SecsTo1970 = 116444736000000000;

    lgTemp = Int32x32To64(pevlr->TimeGenerated,10000000) + SecsTo1970;

    FileTime.dwLowDateTime = (DWORD) lgTemp;
    FileTime.dwHighDateTime = (DWORD)(lgTemp >> 32);

    FileTimeToLocalFileTime(&FileTime, &LocalFileTime);
    FileTimeToSystemTime(&LocalFileTime, &SysTime);

    printf("Time Generated: %02d/%02d/%02d   %02d:%02d:%02d\n",
        SysTime.wMonth,
        SysTime.wDay,
        SysTime.wYear,
        SysTime.wHour,
        SysTime.wMinute,
        SysTime.wSecond);
}
```

## Event Logging Reference

The following elements are used with event logging.

- [Event Logging Functions](#)
- [Event Logging Structures](#)

## Event Logging Functions

The following functions are used with event logging.

[BackupEventLog](#)

[ClearEventLog](#)

[CloseEventLog](#)

[DeregisterEventSource](#)

[GetNumberOfEventLogRecords](#)

[GetOldestEventLogRecord](#)

[NotifyChangeEventLog](#)

[OpenBackupEventLog](#)

[OpenEventLog](#)

[ReadEventLog](#)

[RegisterEventSource](#)

[ReportEvent](#)

## BackupEventLog

The **BackupEventLog** function saves the specified event log to a backup file. The function does not clear the event log.

```
BOOL BackupEventLog(  
    HANDLE hEventLog,           // handle to event log  
    LPCTSTR lpBackupFileName    // name of backup file  
);
```

### Parameters

*hEventLog*

Handle to the open event log. This handle is returned by the [OpenEventLog](#) or [OpenBackupEventLog](#) function.

*lpBackupFileName*

Pointer to a null-terminated string that names the backup file.

### Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

### Remarks

The backup file cannot be written to a remote server because this function is implemented by a service running in the LocalSystem account, which does not have credentials on the remote

machine. However, it is possible to write the file to a remote machine using a null session.

## QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT.

## See Also

[Event Logging Overview](#), [Event Logging Functions](#), [OpenBackupEventLog](#), [OpenEventLog](#)

# ClearEventLog

The **ClearEventLog** function clears the specified event log, and optionally saves the current copy of the logfile to a backup file.

```
BOOL ClearEventLog(  
    HANDLE hEventLog,           // handle to event log  
    LPCTSTR lpBackupFileName   // name of backup file  
);
```

## Parameters

### *hEventLog*

Handle to the event log to be cleared. This handle is returned by the [OpenEventLog](#) function.

### *lpBackupFileName*

Pointer to the null-terminated string specifying the name of a file in which a current copy of the event logfile will be placed. If this file already exists, the function fails.

If the *lpBackupFileName* parameter is NULL, the current event logfile is not backed up.

## Return Values

If the function succeeds, the return value is nonzero. The specified event log has been backed up (if *lpBackupFileName* is not NULL) and then cleared.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). The **ClearEventLog** function fails if the event log is empty or a file already exists with the same name as *lpBackupFileName*.

## Remarks

After this function returns, any handles that reference the cleared event log cannot be used to read the log.

The **ClearEventLog** function is used to optionally back up an existing logfile of the module represented by *hEventLog*. The function backs up the logfile to another file, and then clears the existing logfile. The caller must have write permission for the path specified, and must also have permission to clear the current logfile.

### QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT.

### See Also

[Event Logging Overview](#), [Event Logging Functions](#), [OpenEventLog](#)

## CloseEventLog

The **CloseEventLog** function closes the specified event log.

```
BOOL CloseEventLog(  
    HANDLE hEventLog    // handle to event log  
);
```

### Parameters

*hEventLog*

Handle to the event log to be closed. This handle is returned by the **OpenEventLog** function.

### Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

### QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

### See Also

[Event Logging Overview](#), [Event Logging Functions](#), [OpenEventLog](#)

## DeregisterEventSource

The **DeregisterEventSource** function closes a handle to the specified event log.

```
BOOL DeregisterEventSource(  
    HANDLE hEventLog    // handle to event log  
);
```

### Parameters

*hEventLog*

Handle to the event log. This handle is returned by the [RegisterEventSource](#) function.

### Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

### QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

### See Also

[Event Logging Overview](#), [Event Logging Functions](#), [RegisterEventSource](#)

## GetNumberOfEventLogRecords

The **GetNumberOfEventLogRecords** function retrieves the number of records in the specified event log.

```
BOOL GetNumberOfEventLogRecords(  
    HANDLE hEventLog,          // handle to event log  
    PDWORD NumberOfRecords    // buffer for number of records  
);
```

### Parameters

*hEventLog*

Handle to the open event log. This handle is returned by the [OpenEventLog](#) or [OpenBackupEventLog](#) function.

*NumberOfRecords*

Pointer to the buffer that receives the number of records in the specified event log.

**Return Values**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

**QuickInfo**

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

**See Also**

[Event Logging Overview](#), [Event Logging Functions](#), [GetOldestEventLogRecord](#), [OpenBackupEventLog](#), [OpenEventLog](#)

## GetOldestEventLogRecord

The **GetOldestEventLogRecord** function retrieves the absolute record number of the oldest record in the specified event log.

```
BOOL GetOldestEventLogRecord(  
    HANDLE hEventLog,        // handle to event log  
    PDWORD OldestRecord     // buffer for number of oldest record  
);
```

**Parameters***hEventLog*

Handle to the open event log. This handle is returned by the [OpenEventLog](#) or [OpenBackupEventLog](#) function.

*OldestRecord*

Pointer to the buffer that receives the absolute record number of the oldest record in the specified event log.

**Return Values**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

### QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

### See Also

[Event Logging Overview](#), [Event Logging Functions](#), [GetNumberOfEventLogRecords](#), [OpenBackupEventLog](#), [OpenEventLog](#)

## NotifyChangeEventLog

The **NotifyChangeEventLog** function enables an application to receive notification when an event is written to the specified event log file. When the event is written to the event log file, the specified event object is set to the signaled state.

```
BOOL NotifyChangeEventLog(  
    HANDLE hEventLog, // handle to an event log  
    HANDLE hEvent     // handle to an event object  
);
```

### Parameters

*hEventLog*

Handle to an event log file. This handle is returned by the [OpenEventLog](#) or [OpenBackupEventLog](#) function.

*hEvent*

Handle to an event object. Use the [CreateEvent](#) function to create the event object.

### Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

### Remarks

The **NotifyChangeEventLog** function does not work with remote handles. If the *hEventLog* parameter is the handle to an event log on a remote computer, **NotifyChangeEventLog** returns zero, and **GetLastError** returns ERROR\_INVALID\_HANDLE.

When an event is written to the log file specified by *hEventLog*, the event specified by the *hEvent* parameter is set to the signaled state.

### QuickInfo

**Windows NT:** Requires version 3.5 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

### See Also

[Event Logging Overview](#), [Event Logging Functions](#), [CreateEvent](#), [OpenBackupEventLog](#), [OpenEventLog](#)

## OpenBackupEventLog

The **OpenBackupEventLog** function opens a handle to a backup event log. This handle can be used with the [BackupEventLog](#) function.

```
HANDLE OpenBackupEventLog(  
    LPCTSTR lpUNCServerName, // backup file server name  
    LPCTSTR lpFileName       // backup filename  
);
```

### Parameters

*lpUNCServerName*

Pointer to a null-terminated string that specifies the Universal Naming Convention (UNC) name of the server on which this operation is to be performed.

*lpFileName*

Pointer to a null-terminated string that specifies the name of the backup file.

### Return Values

If the function succeeds, the return value is a handle to the backup event log.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

### Remarks

The backup file cannot be written to a remote server because this function is implemented by a service running in the LocalSystem account, which does not have credentials on the remote machine. However, it is possible to write the file to a remote machine using a null session.

If the backup filename specifies a remote server, *lpUNCServerName* must be NULL.

## QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT.

## See Also

[Event Logging Overview](#), [Event Logging Functions](#), [BackupEventLog](#)

# OpenEventLog

The **OpenEventLog** function opens a handle to an event log.

```
HANDLE OpenEventLog(  
    LPCTSTR lpUNCServerName, // pointer to server name  
    LPCTSTR lpSourceName    // pointer to source name  
);
```

## Parameters

*lpUNCServerName*

Pointer to a null-terminated string that specifies the Universal Naming Convention (UNC) name of the server on which the event log is to be opened.

*lpSourceName*

Pointer to a null-terminated string that specifies the name of the logfile that the returned handle will reference. This can be the Application, Security, or System logfile, or a custom registered logfile. If a custom registered logfile name cannot be found, the event logging service opens the Application logfile, however, there will be no associated message or category string file.

## Return Values

If the function succeeds, the return value is the handle to an event log.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

## QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT.

## See Also

[Event Logging Overview](#), [Event Logging Functions](#), [ClearEventLog](#), [CloseEventLog](#), [GetNumberOfEventLogRecords](#), [GetOldestEventLogRecord](#), [ReadEventLog](#), [ReportEvent](#)

# ReadEventLog

The **ReadEventLog** function reads a whole number of entries from the specified event log. The function can be used to read log entries in forward or reverse chronological order.

```

BOOL ReadEventLog(
    HANDLE hEventLog,           // handle to event log
    DWORD dwReadFlags,         // specifies how to read log
    DWORD dwRecordOffset,      // number of first record
    LPVOID lpBuffer,           // address of buffer for read data
    DWORD nNumberOfBytesToRead, // number of bytes to read
    DWORD *pnBytesRead,        // number of bytes read
    DWORD *pnMinNumberOfBytesNeeded // number of bytes required for next
                                   // record
);

```

## Parameters

### *hEventLog*

Handle to the event log to read. This handle is returned by the **OpenEventLog** function.

### *dwReadFlags*

Specifies how the read operation is to proceed. This parameter can be any combination of the following values:

Value	Meaning
EVENTLOG_FORWARDS_READ	The log is read in forward chronological order.
EVENTLOG_BACKWARDS_READ	The log is read in reverse chronological order.
EVENTLOG_SEEK_READ	The read operation proceeds from the record specified by the <i>dwRecordOffset</i> parameter. If this flag is used, <i>dwReadFlags</i> must also specify <b>EVENTLOG_FORWARDS_READ</b> or <b>EVENTLOG_BACKWARDS_READ</b> . If the buffer is large enough, more than one record can be read at the specified seek position; the additional flag indicates the direction for successive read operations.
EVENTLOG_SEQUENTIAL_READ	The read operation proceeds sequentially from the last call to the <b>ReadEventLog</b> function using this handle.

*dwRecordOffset*

Specifies the log-entry record number at which the read operation should start. This parameter is ignored unless the *dwReadFlags* parameter includes the `EVENTLOG_SEEK_READ` flag.

*lpBuffer*

Pointer to a buffer for the data read from the event log. This parameter cannot be `NULL`, even if the *nNumberOfBytesToRead* parameter is zero.

The buffer will be filled with an [EVENTLOGRECORD](#) structure.

*nNumberOfBytesToRead*

Specifies the size, in bytes, of the buffer. This function will read as many whole log entries as will fit in the buffer; the function will not return partial entries, even if there is room in the buffer.

*pnBytesRead*

Pointer to a variable that receives the number of bytes read by the function.

*pnMinNumberOfBytesNeeded*

Pointer to a variable that receives the number of bytes required for the next log entry. This count is valid only if **ReadEventLog** returns zero and **GetLastError** returns `ERROR_INSUFFICIENT_BUFFER`.

**Return Values**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

**Remarks**

When this function returns successfully, the read position in the error log is adjusted by the number of records read. Only a whole number of event log records will be returned.

**Note** The configured filename for this source may also be the configured filename for other sources (several sources can exist as subkeys under a single logfile). Therefore, this function may return events that were logged by more than one source.

**QuickInfo**

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in `winbase.h`.

**Import Library:** Use `advapi32.lib`.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT.

**See Also**

[Event Logging Overview](#), [Event Logging Functions](#), [ClearEventLog](#), [CloseEventLog](#), [EVENTLOGRECORD](#), [OpenEventLog](#), [ReportEvent](#)

# RegisterEventSource

The **RegisterEventSource** function returns a registered handle to an event log.

```
HANDLE RegisterEventSource(  
    LPCTSTR lpUNCServerName, // server name for source  
    LPCTSTR lpSourceName    // source name for registered handle  
);
```

## Parameters

### *lpUNCServerName*

Pointer to a null-terminated string that specifies the Universal Naming Convention (UNC) name of the server on which this operation is to be performed. If this parameter is NULL, the operation is performed on the local computer.

### *lpSourceName*

Pointer to a null-terminated string that specifies the name of the source referenced by the returned handle. The source name must be a subkey of a logfile entry under the **EventLog** key in the registry. For example, WinApp is a valid source name if the registry has the following key:

```
HKEY_LOCAL_MACHINE  
    System  
        CurrentControlSet  
            Services  
                EventLog  
                    Application  
                        WinApp  
                    Security  
                        System
```

## Return Values

If the function succeeds, the return value is a handle that can be used with the [ReportEvent](#) function.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

## Remarks

If the source name cannot be found, the event logging service uses the **Application** logfile; it does not create a new source. Events are reported for the source, however, there are no message and category message files specified for looking up descriptions of the event identifiers for the source.

To close the handle to the event log, call the [DeregisterEventSource](#) function.

## QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT.

## See Also

[Event Logging Overview](#), [Event Logging Functions](#), [DeregisterEventSource](#), [ReportEvent](#)

# ReportEvent

The **ReportEvent** function writes an entry at the end of the specified event log.

```

BOOL ReportEvent(
    HANDLE hEventLog,      // handle returned by RegisterEventSource
    WORD wType,           // event type to log
    WORD wCategory,       // event category
    DWORD dwEventID,      // event identifier
    PSID lpUserSid,       // user security identifier (optional)
    WORD wNumStrings,     // number of strings to merge with message
    DWORD dwDataSize,     // size of binary data, in bytes
    LPCTSTR *lpStrings,   // array of strings to merge with message
    LPVOID lpRawData      // address of binary data
);

```

## Parameters

### *hEventLog*

Handle to the event log. This handle is returned by the [RegisterEventSource](#) function.

### *wType*

Specifies the type of event being logged. This parameter can be one of the following values:

Value	Meaning
EVENTLOG_ERROR_TYPE	Error event
EVENTLOG_WARNING_TYPE	Warning event
EVENTLOG_INFORMATION_TYPE	Information event
EVENTLOG_AUDIT_SUCCESS	Success Audit event
EVENTLOG_AUDIT_FAILURE	Failure Audit event

For more information about event types, see [Event Types](#).

### *wCategory*

Specifies the event category. This is source-specific information; the category can have any value.

### *dwEventID*

Specifies the event. The event identifier specifies the message that goes with this event as an entry in the message file associated with the event source.

### *lpUserSid*

Pointer to the current user's security identifier. This parameter can be NULL if the security identifier is not required.

### *wNumStrings*

Specifies the number of strings in the array pointed to by the *lpStrings* parameter. A value

of zero indicates that no strings are present.

#### *dwDataSize*

Specifies the number of bytes of event-specific raw (binary) data to write to the log. If this parameter is zero, no event-specific data is present.

#### *lpStrings*

Pointer to a buffer containing an array of null-terminated strings that are merged into the message from the message file before Event Viewer displays the string to the user. This parameter must be a valid pointer (or NULL), even if *wNumStrings* is zero. Each string has a limit of 32K bytes.

#### *lpRawData*

Pointer to the buffer containing the binary data. This parameter must be a valid pointer (or NULL), even if the *dwDataSize* parameter is zero.

## Return Values

If the function succeeds, the return value is nonzero, indicating that the entry was written to the log.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

## Remarks

This function is used to log an event. The entry is written to the end of the configured logfile for the source identified by the *hEventLog* parameter. The **ReportEvent** function adds the time, the entry's length, and the offsets before storing the entry in the log. To enable the function to add the username, you must supply the user's SID in the *lpUserSid* parameter.

## QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winbase.h.

**Import Library:** Use advapi32.lib.

**Unicode:** Implemented as Unicode and ANSI versions on Windows NT.

## See Also

[Event Logging Overview](#), [Event Logging Functions](#), [ClearEventLog](#), [CloseEventLog](#), [OpenEventLog](#), [ReadEventLog](#), [RegisterEventSource](#)

## Event Logging Structures

The following structure is used with event logging.

### [EVENTLOGRECORD](#)

# EVENTLOGRECORD

The **EVENTLOGRECORD** structure contains information about an event record returned by the [ReadEventLog](#) function.

```
typedef struct _EVENTLOGRECORD {
    DWORD Length;
    DWORD Reserved;
    DWORD RecordNumber;
    DWORD TimeGenerated;
    DWORD TimeWritten;
    DWORD EventID;
    WORD EventType;
    WORD NumStrings;
    WORD EventCategory;
    WORD ReservedFlags;
    DWORD ClosingRecordNumber;
    DWORD StringOffset;
    DWORD UserSidLength;
    DWORD UserSidOffset;
    DWORD DataLength;
    DWORD DataOffset;
    //
    // Then follow:
    //
    // TCHAR SourceName[ ]
    // TCHAR Computername[ ]
    // SID UserSid
    // TCHAR Strings[ ]
    // BYTE Data[ ]
    // CHAR Pad[ ]
    // DWORD Length;
    //
} EVENTLOGRECORD;
```

## Members

### Length

Specifies the length, in bytes, of this event record. Note that this value is stored at both ends of the entry to ease moving forward or backward through the log. The length includes any pad bytes inserted at the end of the record for DWORD alignment.

### Reserved

Reserved.

### RecordNumber

Contains a record number that can be used with the `EVENTLOG_SEEK_READ` flag passed in a call to the [ReadEventLog](#) function to begin reading at a specified record.

### TimeGenerated

The time at which this entry was submitted. This time is measured in the number of seconds elapsed since 00:00:00 January 1, 1970, Universal Coordinated Time.

### TimeWritten

Specifies the time at which this entry was received by the service to be written to the logfile. This time is measured in the number of seconds elapsed since 00:00:00 January 1, 1970, Universal Coordinated Time.

### EventID

Specifies the event. This is specific to the source that generated the event log entry, and is used, together with **SourceName**, to identify a message in a message file that is presented to the user while viewing the log.

**EventType**

Specifies the type of event. This member can be one of the following values:

<b>Value</b>	<b>Meaning</b>
EVENTLOG_ERROR_TYPE	Error event
EVENTLOG_WARNING_TYPE	Warning event
EVENTLOG_INFORMATION_TYPE	Information event
EVENTLOG_AUDIT_SUCCESS	Success Audit event
EVENTLOG_AUDIT_FAILURE	Failure Audit event

For more information about event types, see [Event Logging](#).

**NumStrings**

Specifies the number of strings present in the log (at the position indicated by **StringOffset**). These strings are merged into the message before it is displayed to the user.

**EventCategory**

Specifies a subcategory for this event. This subcategory is source specific.

**ReservedFlags**

Reserved.

**ClosingRecordNumber**

Reserved.

**StringOffset**

Specifies the offset of the strings within this event log entry.

**UserSidLength**

Specifies the length, in bytes, of the **UserSid** member. This value can be zero if no security identifier was provided.

**UserSidOffset**

Specifies the offset of the security identifier (SID) within this event record. To obtain the user name for this SID, use the [LookupAccountSid](#) function.

**DataLength**

Specifies the length, in bytes, of the event-specific data (at the position indicated by **DataOffset**).

**DataOffset**

Specifies the offset of the event-specific information within this event record. This information could be something specific (a disk driver might log the number of retries, for example), followed by binary information specific to the event being logged and to the source that generated the entry.

**SourceName**

Contains the variable-length null-terminated string specifying the name of the source (application, service, driver, subsystem) that generated the entry. This is the name used to retrieve from the registry the name of the file containing the message strings for this source. It is used, together with the event identifier, to get the message string that describes this event.

**Computername**

Contains the variable-length null-terminated string specifying the name of the computer that generated this event. There may also be some pad bytes after this field to ensure that the **UserSid** is aligned on a DWORD boundary.

**UserSid**

Specifies the security identifier of the active user at the time this event was logged. This member may be empty if the **UserSidLength** member is zero.

The defined members are followed by the replacement strings for the message identified by the event identifier, the binary information, some pad bytes to make sure the full entry is on a

**DWORD** boundary, and finally the length of the log entry again. Because the strings and the binary information can be of any length, no structure members are defined to reference them.

The event identifier together with **SourceName** and a language identifier identify a message string that describes the event in more detail. The strings are used as replacement strings and are merged into the message string to make a complete message. The message strings are contained in a message file specified in the source entry in the registry. To obtain the appropriate message string from the message file, load the message file with the [LoadLibrary](#) function and use the [FormatMessage](#) function.

The binary information is information that is specific to the event. It could be the contents of the processor registers when a device driver got an error, a dump of an invalid packet that was received from the network, a dump of all the structures in a program (when the data area was detected to be corrupt), and so on. This information should be useful to the writer of the device driver or the application in tracking down bugs or unauthorized breaks into the application.

### QuickInfo

**Windows NT:** Requires version 3.1 or later.

**Windows:** Unsupported.

**Windows CE:** Unsupported.

**Header:** Declared in winnt.h.

### See Also

[Event Logging Overview](#), [Event Logging Structures](#), [LookupAccountSid](#), [ReadEventLog](#)