Legal Information

Performance Data Helper DLL

Performance Data Helper

You can use the *performance data* provided by applications, services, and drivers to determine system bottlenecks and fine-tune system and application performance. The easiest way to display the performance data is to use the Performance Monitor application located in the Administrative Tools group.

If you need to collect performance data for your application, the easiest way to do this is to use the interface provided by the Performance Data Helper (PDH) interface. Applications that need more control over performance data collection can use the registry interface directly. This is the method that is used by the functions in PDH.DLL and by the Performance Monitor. It is more efficient for the Performance Monitor to use the registry interface, because it displays counters grouped by object. If you are retrieving individual counters, rather than a group of counters from a particular object, it is just as efficient to use the PDH interface.

This overview discusses both interfaces.

- The PDH Interface
- The Registry Interface

In addition, this overview discusses how to add your own performance counters, by creating an extensible performance DLL. For more information, see Adding Performance Counters.

The PDH Interface

- Collecting Performance Data.
- Checking PDH Return Values
- Enumerating Process Objects Using PDH

Collecting Performance Data

PDH functions work with queries and counters. A *query* is a set of performance counters that are grouped together so that you can collect their data at the same time. A *counter* is a performance data item stored in the Windows NT Performance Registry. You can call the **PdhBrowseCounters** function to display the following PDH dialog box, which lets you browse the available counters.

{ewc msdncd, EWGraphic, bsd23561 0 /a "SDK.WMF"}

For more information on counters, see Performance Objects and Counters.

A query can contain one or more counters, and a counter can be contained in more than one query. To use PDH, you must create queries and add counters to them, as shown in the following sections.

- Creating a query
- Collecting data
- Displaying data
- Ending data collection

Creating a Query

To create a new query, call the **PdhOpenQuery** function. The function returns a handle to the query to be used in other PDH function calls. You can create multiple queries to be used in your program.

After you create a query, call the **PdhAddCounter** function to add a counter to the query. Call **PdhAddCounter** for each counter to be added to the query. You can use one of the following methods to determine which counters to use.

- Hard-code the counter name in the call to **PdhAddCounter**. Use this method if you always monitor the same counter.
- Read the name from the registry or an initialization file. Use this method if the set of counters you monitor can change.
- Call the **PdhBrowseCounters** function to display the PDH browse dialog box. The dialog box allows the user to browse and select performance counters. The function returns one or more counter path strings for use in a query. If you add the counter returned by **PdhBrowseCounters** to a query, you can use the **PdhRemoveCounter** function to remove the counter from the query.

Counter Paths

The counter names used by PDH are expressed as counter paths. A *counter path* uses a collection of hierarchical elements to describe a counter. In the same way that a file path includes drives, directories, subdirectories, and file names, a counter path includes machines, objects, instances, and counter names. The syntax for a counter path is:

\\Machine\PerfObject(ParentInstance/ObjectInstance#InstanceIndex)\Counter

The \\Machine component is optional; it specifies the name of the machine. If you do not supply this component, PDH uses the local machine name. Therefore, a single path string works on any machine that supports the counter.

The \PerfObject component is required; it specifies the performance object that contains the performance counter. These objects are listed in the PDH browse dialog box and in Performance Monitor **Add To Chart** dialog box, in the **Objects** combo box. If this object has a variable list of instances, then you must also specify an instance string.

The (ParentInstance/ObjectInstance#InstanceIndex) component is only required if the object supports multiple instances. If the instance supports a variable list of instances, then you must specify an instance string. The format of the string depends on the object type. If the object has simple instances, then the format is just the instance name enclosed in parentheses. For example:

```
(Explorer)
```

If the instance of this object requires a parent instance name as well, then the parent instance name must come before the object instance, and be separated by a forward slash character. For example:

```
(Explorer/0)
```

If the object has multiple instances that have the same name string, they can be indexed sequentially by specifying the instance index prefixed by a pound sign. Instance indexes are 0-based, so all instances have an implicit "0" index. For example:

```
(Explorer/0#1)
```

The \Counter component is required; it specifies the performance counter. The counter names are displayed in the PDH browse dialog box and in the Performance Monitor **Add To Chart** dialog box, in the

Counters list box. The counters associated with the selected object are displayed.

The following are two common counters in the PDH counter path format:

```
\Processor(0)\% Processor Time
\System\% Total Processor Time
```

Collecting Data

After you have created a query and added counters to it, call the **PdhCollectQueryData** function to retrieve the current raw data for all counters in the query. Many counters, such as rate counters, require the following two data samples before a valid data value can be determined.

- Call **PdhCollectQueryData** before entering the section of code that you are measuring, to provide a starting value.
- Call **PdhCollectQueryData** again after exiting the section of code that you are measuring, to get the ending value.

Displaying Data

The raw value of many counters is not what the user would expect to see. To get the information in the form described by most counter names, you need to compute the displayable value based on this raw data. PDH does this for you. For example, the Page Faults/Sec counter simply counts page faults, so if you view the raw data, you see a running total of page faults. To get the rate implied by the counter name, page faults per second, you need to call the **PdhGetFormattedCounterValue** function. In this case, the function divides the difference between two samples by the time between the samples.

The **PdhGetFormattedCounterValue** function performs its computations on the most recent sample. If you need to recalculate and redisplay a sample, you must store the raw data and then call the **PdhCalculateCounterFromRawValue** function to perform the calculation on the stored data.

Displaying Statistical Data

Calculating statistical values can be difficult, because many counter values are the result of a change over time. The PDH provides functions that perform calculations (such as minimum, maximum, and average values) on the raw counter data for you.

After you have retrieved the raw data for a counter, call the **PdhComputeCounterStatistics** function to perform the necessary calculations on the raw data.

Ending Data Collection

After you are finished collecting data for a query, call the **PdhCloseQuery** function to close the query and release all allocated system resources. The function closes all counter handles associated with the query.

Checking PDH Return Values

The return value of PDH functions indicates the success or error of the function call, which is distinct from the status of the counter data. Always check the **CStatus** member of a counter value returned in the PDH structures to ensure that the data returned is valid before you use it. If the value of the **CStatus** member does not indicate success, do not use the data. The following are the possible status values for counters:

PDH_CSTATUS_NO_MACHINE

PDH was unable to connect to the machine specified in the counter path. If this status is returned when the counter is being added, the counter is not completely initialized. Each time the query is updated, PDH retries the connection. When the connection is established, normal data collection resumes.

PDH_CSTATUS_NO_OBJECT

The specified machine was found, but no matching performance object was found on that machine. If this status is returned when the counter is being added, the specified counter is not included in the query. If this status is returned by an active counter, the data for that counter is invalid. Each time the data is requested, PDH tries to obtain this counter data.

PDH_CSTATUS_NO_INSTANCE

The specified instance was not found in the object. If this status is returned while the counter is being added to the query, the counter is successfully added to the query, but no data is available until the specific instance appears and a successful status is returned.

PDH_CSTATUS_NO_COUNTER

The specified counter was not found in the specified object. If this status is returned when the counter is being added, then the counter is not added to the query. If this status is returned after a data update, the data for that counter is invalid. Each time the data is requested, PDH tries to obtain this counter data.

PDH_CSTATUS_INVALID_DATA

The counter was successfully found, but the data returned is not valid. One possible cause for this value is when the data of a normally increasing counter is less than the previous value. Another possible cause is a system timer that is not correct.

PDH_CSTATUS_VALID_DATA

The data for the counter was returned successfully, but is unchanged from the last time the counter was read.

PDH_CSTATUS_NEW_DATA

The data for the counter was returned successfully and is different from the last time the counter was read. PDH_CSTATUS_NEW_DATA can be returned on a rate counter even if the resulting rate is the same as the last sample. This is because the raw data value that is used in the determination of this status value has changed, not the computed rate.

PDH_CSTATUS_NO_COUNTERNAME

No counter path was specified.

PDH_CSTATUS_BAD_COUNTERNAME

The counter path format is incorrect.

Enumerating Process Objects Using PDH

The example in this topic uses the PDH interface to enumerate the process objects on the system.

```
#ifdef UNICODE
#ifndef _UNICODE
#define _UNICODE  1
#endif
#define tmain     wmain
#else
#define tmain     main
#endif

// This program only needs the essential windows header files.
#define WIN32_LEAN_AND_MEAN 1

#include <windows.h>
#include <winperf.h>
#include <malloc.h>
#include <stdio.h>
#include <tchar.h>
#include <pdh.h>

int
tmain ()
{
    PDH_STATUS  pdhStatus               = ERROR_SUCCESS;
    LPTSTR      szCounterListBuffer     = NULL;
    DWORD       dwCounterListSize       = 0;
    LPTSTR      szInstanceListBuffer    = NULL;
    DWORD       dwInstanceListSize      = 0;
    LPTSTR      szThisInstance          = NULL;


// Determine the required buffer size for the data.

    pdhStatus = PdhEnumObjectItems (
        NULL,                   // reserved
        NULL,                   // local machine
        TEXT("Process"),        // object to enumerate
        szCounterListBuffer,    // pass in NULL buffers
        &dwCounterListSize,     // an 0 length to get
        szInstanceListBuffer,   // required size
        &dwInstanceListSize,    // of the buffers in chars
        PERF_DETAIL_WIZARD,     // counter detail level
        0);

    if (pdhStatus == ERROR_SUCCESS)
    {
    // Allocate the buffers and try the call again.
        szCounterListBuffer = (LPTSTR)malloc (
            (dwCounterListSize * sizeof (TCHAR)));
        szInstanceListBuffer = (LPTSTR)malloc (
            (dwInstanceListSize * sizeof (TCHAR)));
```

```
        if ((szCounterListBuffer != NULL) &&
            (szInstanceListBuffer != NULL))
        {
            pdhStatus = PdhEnumObjectItems (
                NULL,   // reserved
                NULL,   // local machine
                TEXT("Process"), // object to enumerate
                szCounterListBuffer,    // pass in NULL buffers
                &dwCounterListSize,     // an 0 length to get
                szInstanceListBuffer,   // required size
                &dwInstanceListSize,    // of the buffers in chars
                PERF_DETAIL_WIZARD,     // counter detail level
                0);
            if (pdhStatus == ERROR_SUCCESS)
            {
                _tprintf (TEXT("\nRunning Processes:"));
            // Walk the return instance list.
                for (szThisInstance = szInstanceListBuffer;
                     *szThisInstance != 0;
                     szThisInstance += lstrlen(szThisInstance) + 1)
                {
                     _tprintf (TEXT("\n  %s"), szThisInstance);
                }
            }
        }
        else
        {
            _tprintf (TEXT("\nPROCLIST: unable to allocate buffers"));
        }

        if (szCounterListBuffer != NULL)
            free (szCounterListBuffer);

        if (szInstanceListBuffer != NULL)
            free (szInstanceListBuffer);
    }
    else
    {
        _tprintf(TEXT("\nUnable to determine required buffer size."));
    }
    return 0;
}
```

The Registry Interface

Performance Objects and Counters

The performance data is grouped by *performance object* to which it is related. A performance object consists of *performance counters*, which are used to measure various aspects of system, application, or device performance. For example, the Processor object includes the Processor Time counter to measure the percentage of time the processor spends executing threads that are not idle and the Interrupts/sec counter to measure the number of device interrupts the processor receives.

An *instance* is a unique copy of a particular object type. Not all object types support multiple instances. For example, a Memory object has only one instance, because a system has one memory. However, the Processor object supports multiple instances, because a system can have one or more processors.

Most performance counters in Windows NT increment and are never cleared. Therefore, to obtain performance data, use the following steps.

- Take a snapshot of the relevant performance counters at the beginning of a time interval.
- Take a snapshot of the same performance counters at the end of the time interval.
- Find the difference between the counter values in the snapshots and apply the appropriate calculation function.

For a list of the system performance objects and performance counters, see Windows NT Performance Counters.

The HKEY_PERFORMANCE_DATA Key

The performance data is accessed through the registry key **HKEY_PEFORMANCE_DATA**. Each software component creates keys for its objects and counters when it is installed and writes counter data while it is executing. You can access this data as you would access any other registry data, using the registry functions. However, although you use the registry to collect performance data, the data is not stored in the registry database. Instead, calling the registry functions with the **HKEY_PEFORMANCE_DATA** key causes the system to collect the data from the appropriate system object managers.

To obtain performance data from the local system, use the **RegQueryValueEx** function, with the **HKEY_PERFORMANCE_DATA** key. The first call opens the key; you do not need to explicitly open the key first. However, be sure to use the **RegCloseKey** function to close the handle to the key when you are finished obtaining performance data. The user cannot install or remove a software component while its performance data is in use.

To obtain performance data from a remote system, you must use the **RegConnectRegistry** function, with the computer name of the remote system and the **HKEY_PERFORMANCE_DATA** key. This call retrieves a key representing the performance data for the remote system. To retrieve the data, call **RegQueryValueEx** using this key, rather than the **HKEY_PERFORMANCE_DATA** key.

Performance Data Format

The format of the data retrieved by the **RegQueryValueEx** function begins with a single header structure, **PERF_DATA_BLOCK**. The **PERF_DATA_BLOCK** structure describes the system and the performance data. The **PERF_DATA_BLOCK** structure is followed by a list of object information blocks (one per object).

PERF_DATA_BLOCK
Object 1 information
Object 2 information
Object 3 information
…
Object X information

Basic Performance Data Structure

Each object information block contains a **PERF_OBJECT_TYPE** structure, which describes the performance data for the object. The **PERF_OBJECT_TYPE** structure is followed by a list of **PERF_COUNTER_DEFINITION** structures, one for each counter defined for the object. For an object with only one instance, the list of **PERF_COUNTER_DEFINITION** structures is followed by a single **PERF_COUNTER_BLOCK** structure, followed by the data for each counter.

PERF_OBJECT_TYPE
PERF_COUNTER_DEFINITION 1
PERF_COUNTER_DEFINITION 2
PERF_COUNTER_DEFINITION 3
…
PERF_COUNTER_DEFINITION Y
PERF_COUNTER_BLOCK
Counter 1 data
Counter 2 data
Counter 3 data
…
Counter Y data

Structure of Object Information Block (One Instance)

For an object type that supports multiple instances, the list of **PERF_COUNTER_DEFINITION** structures is followed by a list of instance information blocks (one for each instance).

PERF_OBJECT_TYPE
PERF_COUNTER_DEFINITION 1
PERF_COUNTER_DEFINITION 2
PERF_COUNTER_DEFINITION 3

…

PERF_COUNTER_DEFINITIO N Y

Instance 1 Information

Instance 2 Information

Instance 3 Information

…

Instance Z Information

Structure of Object Information Block (Multiple Instances)

Each instance information block contains a **PERF_INSTANCE_DEFINITION** structure and a **PERF_COUNTER_BLOCK** structure.

PERF_INSTANCE_DEFINIT ION

Instance name

PERF_COUNTER_BLOCK

Counter Data 1

Counter Data 2

Counter Data 3

…

Counter Data Y

Structure of Instance Information Block

Navigating the Performance Data

The performance data contains information for a variable number of object types, instances per object, and counters per object type. Therefore, the number and size of blocks in the performance data varies. To ensure that your application correctly receives the performance data, you must use the offsets included in the performance structures to navigate through the data. Every offset is a count of bytes relative to the structure containing it.

For an example that navigates the registry, see Displaying Object, Instance, and Counter Names.

Note The reason the system uses offsets instead of pointers is that pointers are not valid across process boundaries. The addresses that the process that installs the counters would store would not be valid for the process that reads the counters.

Retrieving Counter Names and Explanations

Object type names, counter names, object explanations, and counter explanations are not made directly available in the performance data structures. Instead, the performance data structures contain indices you can use to locate where the names and explanations for each object and counter can be found. The **ObjectNameTitleIndex** and **ObjectHelpTitleIndex** members of the **PERF_OBJECT_TYPE** structure contain the indices to the object name and explanation, respectively. The **CounterNameTitleIndex** and **CounterHelpTitleIndex** members of the **PERF_COUNTER_DEFINITION** structure contain the indices to the counter name and explanation, respectively.

To access the names and explanations, read the **Counter** and **Help** values in the following registry key.

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT
\CurrentVersion\Perflib*langid*

The *langid* is the ASCII representation of the 3-digit hexadecimal language identifier. For example, the U.S. English *langid* is 009. In a non-English version of Windows NT, counters are stored in both the native language of the system and in English.

The data is stored as MULTI_SZ strings. This data type consists of a list of Unicode strings, each terminated with UNICODE_NULL. The last string is followed by an additional UNICODE_NULL. The strings are listed in pairs. The first string of each pair is the Unicode string of the index, and the second string is the actual name of the index. The **Counter** data uses only even-numbered indexes, while the **Help** data has odd-numbered indexes. For example, the **Counter** data contains the following object and counter name strings.

```
2     System
4     Memory
6     % Processor Time
```

The **Help** data contains the following counter explanations.

```
3     The System object type includes those counters that...
5     The Memory object type includes those counters that...
7     Processor Time is expressed as a percentage of the...
```

To retrieve a name or explanation for an object type or counter, given its index, an application should perform the following steps.

1. If the system is remote, call the **RegConnectRegistry** function.
2. Use the **RegOpenKeyEx** function to open the registry key containing the name and explanation text.
3. Use the **RegQueryValueEx** function, specifying either **Counter** or **Help** as the name of the value to query.
4. Convert the index into Unicode or ASCII, depending on whether your application is Unicode or ASCII.
5. Search the MULTI_SZ data for the appropriate index.
6. Retrieve the string following the matching index. The string contains the name or explanation.

If you are going to be looking up a number of counters, you should build a table for faster and easier lookup. For an example, see Displaying Object, Instance, and Counter Names.

Retrieving Selected Data

Retrieving the performance data is not without cost to the system, especially in terms of processor and memory requirements. In cases where your application does not need all the performance data, you can use the *lpszValueName* parameter of the **RegQueryValueEx** function to indicate the amount of information to retrieve. The following table lists the values you can specify for *lpszValueName*. Note that the value strings are case-sensitive, and if a string includes more than one word, your words must be separated by a space.

Value	Description
Global	The function returns all data for counters on the local computer, except those included in the **Costly** category.
nnn xx yyy	Each of these is a Unicode string representing the decimal value for an object name index. The function only returns data about the specified object types on the local computer. For example, if your program were to use "2 4", the System and Memory objects would be retrieved. Note that more object types can be returned than are requested. This occurs when the requested object type depends on another object type. For example, processes are needed to identify threads, so information for the Process object is also returned when you request information for the Thread object.
Foreign *ssss*	The string ***ssss*** is the name of a foreign computer, such as a Novell NetWare server or a UNIX system. If your system is capable of collecting data from a foreign computer, the function returns all data for counters on the foreign computer. Try this approach if the **RegConnectRegistry** function fails to connect to a foreign computer.
Foreign *ssss nnn xx yyy*	This format combines **Foreign *ssss*** and ***nnn xx yyy***.
Costly	The function returns data for object types whose data is expensive to collect in terms of processor time or memory usage. Start a worker thread if your application needs to respond to the user during this lengthy data collection.

Calculations for Raw Counter Data

This section lists the types of performance counters and the calculations that the Performance Monitor uses to convert the raw data to the information you would expect from the counter name. The counter types and calculations are defined in WINPERF.H. For the calculations, the following key applies:

Symbol	Definition
N	Numerator
D	Denominator
TB	Time base (indicates the units of raw data if not used in calculation)
B	Base count (number of entries)
N0	Oldest or first sample taken (of the numerator in this case; D0 would be of the denominator)
N1	More recent sample
Nx	Most recent sample in calculation array (for average calculation)

PERF_100NSEC_MULTI_TIMER Timer for when multiple instances are in use, so the result can exceed 100%. The number of instances is in the next counter.

Element	Value
Numerator	`CounterData`
Denominator	`100NsTime`
Time base	`100Ns`
Calculation	`(N1-N0)/(D1-D0)`
Average function	`(Nx-N0)/(Dx-D0)`

PERF_100NSEC_MULTI_TIMER_INV The inverse of the timer for multiple instances (when the object is not in use).

Element	Value
Numerator	`CounterData`
Denominator	`100NsTime`
Time base	`100Ns`
Calculation	`(B-((N1-N0)/(D1-D0)))`
Average function	`(B-((Nx-N0)/(Dx-D0)))`

PERF_100NSEC_TIMER Timer for when the object is in use.

Element	Value
Numerator	`CounterData`
Denominator	`100NsTime`
Time base	`100Ns`
Calculation	`(N1-N0)/(D1-D0)`
Average function	`(Nx-N0)/(Dx-D0)`

PERF_100NSEC_TIMER_INV The inverse of the timer (when the object is not in use).

Element	Value
Numerator	`CounterData`
Denominator	`100NsTime`
Time base	`100Ns`
Calculation	`(1-((N1-N0)/(D1-D0)))`
Average function	`(1-((Nx-N0)/(Dx-D0)))`

PERF_AVERAGE_BASE Used as the denominator in the computation of time or count averages.

Element	Value
Numerator	`N/A`
Denominator	`N/A`
Time base	`N/A`
Calculation	`N/A`
Average function	`N/A`

PERF_AVERAGE_BULK A count which usually gives the bytes per operation when divided by the number of operations.

Element	Value
Numerator	`CounterData`
Denominator	`BaseData`
Time base	`N/A`
Calculation	`(N1-N0)/(D1-D0)`
Average function	`(Nx-N0)/(Dx-D0)`

PERF_AVERAGE_TIMER A timer which usually gives time per operation when divided by the number of operations.

Element	Value
Numerator	`CounterData`
Denominator	`BaseData`
Time base	`PerfFreq`
Calculation	`((N1-N0)/TB)/(D1-D0)`
Average function	`((Nx-N0)/TB)/(Dx-D0)`

PERF_COUNTER_BULK_COUNT Used to count byte transmission rates.

Element	Value
Numerator	`CounterData`
Denominator	`PerfTime`
Time base	`PerfFreq`
Calculation	`(N1-N0)/((D1-D0)/TB)`
Average function	`(Nx-N0)/((Dx-D0)/TB)`

PERF_COUNTER_COUNTER Rate of counts. The most common counter.

Element	Value
Numerator	`CounterData`
Denominator	`PerfTime`
Time base	`PerfFreq`
Calculation	`(N1-N0)/((D1-D0)/TB)`
Average function	`(Nx-N0)/((Dx-D0)/TB)`

PERF_COUNTER_DELTA Difference between two counters.

Element	Value
Numerator	`CounterData`
Denominator	`N/A`
Time base	`N/A`
Calculation	`(N1-N0)`
Average function	`(Nx-N0)/x`

PERF_COUNTER_LARGE_DELTA Difference between two counters.

Element	Value
Numerator	`CounterData`
Denominator	`N/A`
Time base	`N/A`
Calculation	`(N1-N0)`
Average function	`(Nx-N0)/x`

PERF_COUNTER_LARGE_QUEUELEN_TYPE Average count per time interval.

Element	Value
Numerator	`CounterData`
Denominator	`PerfTime`
Time base	`PerfFreq`
Calculation	`(N1-N0)/(D1-D0)`
Average function	`(Nx-N0)/(Dx-D0)`

PERF_COUNTER_LARGE_RAWCOUNT Instantaneous counter value.

Element	Value
Numerator	`CounterData`
Denominator	`N/A`
Time base	`N/A`
Calculation	`(N0)`
Average function	`SUM(N)/x`

PERF_COUNTER_LARGE_RAWCOUNT_HEX Instantaneous counter value, as a hexadecimal number.

Element	Value
Numerator	`CounterData`

Denominator	`N/A`
Time base	`N/A`
Calculation	`(N0)`
Average function	`SUM(N)/x`

PERF_COUNTER_MULTI_BASE Base for MULTI counters.

Element	Value
Numerator	`N/A`
Denominator	`N/A`
Time base	`N/A`
Calculation	`N/A`
Average function	`N/A`

PERF_COUNTER_MULTI_TIMER Timer for when multiple instances are in use, so the result can exceed 100%. The number of instances is in the next counter.

Element	Value
Numerator	`CounterData`
Denominator	`PerfTime`
Time base	`PerfFreq`
Calculation	`(N1-N0)/(D1-D0)`
Average function	`(Nx-N0)/(Dx-D0)`

PERF_COUNTER_MULTI_TIMER_INV The inverse of the timer for multiple instances (when the object is not in use).

Element	Value
Numerator	`CounterData`
Denominator	`PerfTime`
Time base	`PerfFreq`
Calculation	`(B-((N1-N0)/(D1-D0)))`
Average function	`(B-((Nx-N0)/(Dx-D0)))`

PERF_COUNTER_NODATA There is no data for this counter.

Element	Value
Numerator	0
Denominator	N/A
Time base	N/A
Calculation	0
Average function	0

PERF_COUNTER_QUEUELEN_TYPE Average count per time interval.

Element	Value
Numerator	`CounterData`
Denominator	`PerfTime`

Time base	`PerfFreq`
Calculation	`(N1-N0)/(D1-D0)`
Average function	`(Nx-N0)/(Dx-D0)`

PERF_COUNTER_RAWCOUNT Instantaneous counter value.

Element	Value
Numerator	`CounterData`
Denominator	`N/A`
Time base	`N/A`
Calculation	`(N0)`
Average function	`SUM(N)/x`

PERF_COUNTER_RAWCOUNT_HEX Instantaneous counter value, as a hexadecimal number.

Element	Value
Numerator	`CounterData`
Denominator	`N/A`
Time base	`N/A`
Calculation	`(N0)`
Average function	`SUM(N)/x`

PERF_COUNTER_TEXT Indicates the data is not a counter, but is Unicode text.

Element	Value
Numerator	`N/A`
Denominator	`N/A`
Time base	`N/A`
Calculation	`N/A`
Average function	`N/A`

PERF_COUNTER_TIMER The most common timer.

Element	Value
Numerator	`CounterData`
Denominator	`PerfTime`
Time base	`PerfFreq`
Calculation	`(N1-N0)/(D1-D0)`
Average function	`(Nx-N0)/(Dx-D0)`

PERF_COUNTER_TIMER_INV The inverse of the timer (when the object is not in use).

Element	Value
Numerator	`CounterData`
Denominator	`PerfTime`
Time base	`PerfFreq`
Calculation	`(1-((N1-N0)/(D1-D0)))`
Average function	`(1-((Nx-N0)/(Dx-D0)))`

PERF_ELAPSED_TIME The data is the start time of the item being measured. For display, subtract the start time from the snapshot time to yield the elapsed time. The **PerfTime** member of the **PERF_OBJECT_TYPE** structure contains the sample time. Use the **PerfFreq** member of the **PERF_OBJECT_TYPE** structure to convert the time into seconds.

Element	Value
Numerator	`CounterData`
Denominator	`ObjectTime`
Time base	`ObjectFreq`
Calculation	`(D0-N0)/TB`
Average function	`(Dx-N0)/TB`

PERF_RAW_BASE Used as a base. Check that this value is greater than zero before dividing.

Element	Value
Numerator	`N/A`
Denominator	`N/A`
Time base	`N/A`
Calculation	`N/A`
Average function	`N/A`

PERF_RAW_FRACTION Instantaneous value, to be divided by the base.

Element	Value
Numerator	`CounterData`
Denominator	`BaseData`
Time base	`N/A`
Calculation	`(N0/D0)`
Average function	`SUM(N/D)/x`

PERF_SAMPLE_BASE Used as a base. Check that this value is greater than zero before dividing.

Element	Value
Numerator	`N/A`
Denominator	`N/A`
Time base	`N/A`
Calculation	`N/A`
Average function	`N/A`

PERF_SAMPLE_COUNTER A count that is sampled on each sampling interrupt. Must be divided by the base.

Element	Value
Numerator	`CounterData`
Denominator	`PerfTime`
Time base	`PerfFreq`
Calculation	`(N1-N0)/((D1-D0)/TB)`

Average function	`(Nx-N0)/((Dx-D0)/TB)`

PERF_SAMPLE_FRACTION A count that is either 1 or 0 on each sampling interrupt.

Element	Value
Numerator	`CounterData`
Denominator	`BaseData`
Time base	`N/A`
Calculation	`(N1-N0)/(D1-D0)`
Average function	`(Nx-N0)/(Dx-D0)`

Displaying Object, Instance, and Counter Names

The following example displays the index and name of each object, along with the indices and names of its counters.

The object and counter names are stored in the registry, by index. This example creates a function, GetNameStrings, to load the indices and names of each object and counter from the registry into an array, so that they can be easily accessed. GetNameStrings uses the following standard registry functions to access the data: **RegOpenKey**, **RegCloseKey**, **RegQueryInfoKey**, and **RegQueryValueEx**.

This example creates the following functions for navigating the performance data: FirstObject, FirstInstance, FirstCounter, NextCounter, NextInstance, and NextCounter. These functions navigate the performance data by using the offsets stored in the performance structures.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

#define TOTALBYTES    8192
#define BYTEINCREMENT 1024

LPSTR lpNameStrings;
LPSTR *lpNamesArray;

/*****************************************************************
 *                                                               *
 * Functions used to navigate through the performance data.      *
 *                                                               *
 *****************************************************************/

PPERF_OBJECT_TYPE FirstObject( PPERF_DATA_BLOCK PerfData )
{
    return( (PPERF_OBJECT_TYPE)((PBYTE)PerfData +
        PerfData->HeaderLength) );
}

PPERF_OBJECT_TYPE NextObject( PPERF_OBJECT_TYPE PerfObj )
{
    return( (PPERF_OBJECT_TYPE)((PBYTE)PerfObj +
        PerfObj->TotalByteLength) );
}

PPERF_INSTANCE_DEFINITION FirstInstance( PPERF_OBJECT_TYPE PerfObj )
{
    return( (PPERF_INSTANCE_DEFINITION)((PBYTE)PerfObj +
        PerfObj->DefinitionLength) );
}

PPERF_INSTANCE_DEFINITION NextInstance(
    PPERF_INSTANCE_DEFINITION PerfInst )
{
    PPERF_COUNTER_BLOCK PerfCntrBlk;
```

```
    PerfCntrBlk = (PPERF_COUNTER_BLOCK)((PBYTE)PerfInst +
        PerfInst->ByteLength);

    return( (PPERF_INSTANCE_DEFINITION)((PBYTE)PerfCntrBlk +
        PerfCntrBlk->ByteLength) );
}

PPERF_COUNTER_DEFINITION FirstCounter( PPERF_OBJECT_TYPE PerfObj )
{
    return( (PPERF_COUNTER_DEFINITION) ((PBYTE)PerfObj +
        PerfObj->HeaderLength) );
}

PPERF_COUNTER_DEFINITION NextCounter(
    PPERF_COUNTER_DEFINITION PerfCntr )
{
    return( (PPERF_COUNTER_DEFINITION)((PBYTE)PerfCntr +
        PerfCntr->ByteLength) );
}

/*****************************************************************
 *                                                               *
 * Load the counter and object names from the registry to the    *
 * global variable lpNamesArray.                                 *
 *                                                               *
 *****************************************************************/

void GetNameStrings( )
{
    HKEY hKeyPerflib;      // handle to registry key
    HKEY hKeyPerflib009;   // handle to registry key
    DWORD dwMaxValueLen;   // maximum size of key values
    DWORD dwBuffer;        // bytes to allocate for buffers
    DWORD dwBufferSize;    // size of dwBuffer
    LPSTR lpCurrentString; // pointer for enumerating data strings
    DWORD dwCounter;       // current counter index

// Get the number of Counter items.

    RegOpenKeyEx( HKEY_LOCAL_MACHINE,
        "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Perflib",
        0,
        KEY_READ,
        &hKeyPerflib);

    dwBufferSize = sizeof(dwBuffer);

    RegQueryValueEx( hKeyPerflib,
        "Last Counter",
        NULL,
        NULL,
        (LPBYTE) &dwBuffer,
        &dwBufferSize );

    RegCloseKey( hKeyPerflib );
```

```
// Allocate memory for the names array.

    lpNamesArray = malloc( (dwBuffer+1) * sizeof(LPSTR) );

// Open key containing counter and object names.

    RegOpenKeyEx( HKEY_LOCAL_MACHINE,
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Perflib\\009",
        0,
        KEY_READ,
        &hKeyPerflib009);

// Get the size of the largest value in the key (Counter or Help).

    RegQueryInfoKey( hKeyPerflib009,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        &dwMaxValueLen,
        NULL,
        NULL);

// Allocate memory for the counter and object names.

    dwBuffer = dwMaxValueLen + 1;

    lpNameStrings = malloc( dwBuffer * sizeof(CHAR) );

// Read Counter value.

    RegQueryValueEx( hKeyPerflib009,
        "Counter",
        NULL,
        NULL,
        lpNameStrings, &dwBuffer );

// Load names into an array, by index.

    for( lpCurrentString = lpNameStrings; *lpCurrentString;
         lpCurrentString += (lstrlen(lpCurrentString)+1) )
    {
        dwCounter = atol( lpCurrentString );

        lpCurrentString += (lstrlen(lpCurrentString)+1);

        lpNamesArray[dwCounter] = (LPSTR) lpCurrentString;
    }
}
```

```
/*****************************************************************
 *                                                               *
 * Display the indices and/or names for all performance objects, *
 * instances, and counters.                                      *
 *                                                               *
 *****************************************************************/

void main()
{
    PPERF_DATA_BLOCK PerfData = NULL;
    PPERF_OBJECT_TYPE PerfObj;
    PPERF_INSTANCE_DEFINITION PerfInst;
    PPERF_COUNTER_DEFINITION PerfCntr, CurCntr;
    PPERF_COUNTER_BLOCK PtrToCntr;
    DWORD BufferSize = TOTALBYTES;
    DWORD i, j, k;

// Get the name strings through the registry.

    GetNameStrings( );

// Allocate the buffer for the performance data.

    PerfData = (PPERF_DATA_BLOCK) malloc( BufferSize );

    while( RegQueryValueEx( HKEY_PERFORMANCE_DATA,
                            "Global",
                            NULL,
                            NULL,
                            (LPBYTE) PerfData,
                            &BufferSize ) == ERROR_MORE_DATA )
    {
    // Get a buffer that is big enough.

        BufferSize += BYTEINCREMENT;
        PerfData = (PPERF_DATA_BLOCK) realloc( PerfData, BufferSize );
    }

// Get the first object type.

    PerfObj = FirstObject( PerfData );

// Process all objects.

    for( i=0; i < PerfData->NumObjectTypes; i++ )
    {
    // Display the object by index and name.

        printf( "\nObject %ld: %s\n", PerfObj->ObjectNameTitleIndex,
            lpNamesArray[PerfObj->ObjectNameTitleIndex] );

    // Get the first counter.

        PerfCntr = FirstCounter( PerfObj );
```

```
        if( PerfObj->NumInstances > 0 )
        {
        // Get the first instance.

            PerfInst = FirstInstance( PerfObj );

        // Retrieve all instances.

            for( k=0; k < PerfObj->NumInstances; k++ )
            {
            // Display the instance by name.

                printf( "\n\tInstance %S: \n",
                    (char *)((PBYTE)PerfInst + PerfInst->NameOffset));
                CurCntr = PerfCntr;

            // Retrieve all counters.

                for( j=0; j < PerfObj->NumCounters; j++ )
                {
                // Display the counter by index and name.

                    printf("\t\tCounter %ld: %s\n",
                        CurCntr->CounterNameTitleIndex,
                        lpNamesArray[CurCntr->CounterNameTitleIndex]);

                // Get the next counter.

                    CurCntr = NextCounter( CurCntr );
                }

            // Get the next instance.

                PerfInst = NextInstance( PerfInst );
            }
        }
        else
        {
        // Get the counter block.

            PtrToCntr = (PPERF_COUNTER_BLOCK) ((PBYTE)PerfObj +
                            PerfObj->DefinitionLength );

        // Retrieve all counters.

            for( j=0; j < PerfObj->NumCounters; j++ )
            {
            // Display the counter by index and name.

                printf( "\tCounter %ld: %s\n", PerfCntr-
>CounterNameTitleIndex,
                    lpNamesArray[PerfCntr->CounterNameTitleIndex] );

            // Get the next counter.
```

```
                PerfCntr = NextCounter( PerfCntr );
            }
        }

    // Get the next object type.

        PerfObj = NextObject( PerfObj );
    }
}
```

Adding Performance Counters

Windows NT provides a mechanism for you to add performance objects and counters for your application and other software components. Performance counters specific to your application can help you tune performance while you develop and debug the application. After your application is complete and installed on target systems, the counters can help system administrators adjust configurable settings for your application.

To add an extended object and its counters, use the following steps.

1. Design the object types and counters for the application. See Object and Counter Design.
2. Create an initialization (.INI) file containing the names and descriptions of the counter objects and counters. See Adding Counter Names and Descriptions to the Registry.
3. Create a header (.H) file containing the relative offsets at which the counter objects and counters will be installed in the registry. See Adding Counter Names and Descriptions to the Registry.
4. Set up the necessary performance monitoring entries in the registry. This includes the following steps.
   a. Create a registry key in the **Services** key for the application. If you do not have such a node, create it under the following registry key: **HKEY_LOCAL_MACHINE\SYSTEM\ CurrentControlSet\Services**. Creating the Application's Performance Key
   b. Use the **lodctr** utility with the .INI and .H files to install the information in the registry. This utility succeeds only if a performance key exists in the **Services** key for the application. See Adding Counter Names and Descriptions to the Registry.
5. Create a performance DLL containing a set of exported functions that provide the link between the application and a performance monitoring application. See Creating the Performance DLL.
6. Modify the application's setup file to automate adding information to the registry (as described in step 4) and move your performance DLL to the system directory at setup.

To view an extended object, its functions, and its counter, use the extensible counter list utility (EXCTRLST.EXE). For more information, see ExCtrLst.

Object and Counter Design

A performance object is an entity for which performance data is available. Performance counters define the type of data that is available for a counter object. An application can provide information for multiple performance objects, each with one or more counters.

An application can also define objects that have multiple instances. For example, a SCSI application could use a single set of counter definitions to define a drive object with two counters, such as Bytes Read and Bytes Written. The performance DLL for the application could report performance data for each drive controlled by the application.

The Windows NT Performance Monitor can show some counters as rates, such as Page Faults/sec, as well as a raw count. This gives context to the users, so they do not have to compare data from different time intervals. However, you do not have to take this into account when you design a counter. You increment the counter and let the monitor application do the work of converting raw counts to a rate.

The method you use to collect the data can be as simple as incrementing a counter each time a particular routine in the application is called, or it can involve time-consuming calculations. Counters and timers should increment and never be cleared. Counters can wrap, as long as they do not wrap twice between snapshots. Your application can collect and store data during its normal execution, as long as it does not affect the its performance.

For some types of data, it may be more efficient or appropriate to collect the data on demand. In this situation, the performance DLL must communicate to the application that the data has been requested. For data that is expensive to collect (in terms of processor time or memory usage), consider collecting data only when the performance monitoring program requests **Costly** data. This allows a custom performance monitoring program to routinely request data for all counters that are not costly. The data can be requested only when needed. The Windows NT Performance Monitor does not collect **Costly** data.

Updating the Registry

There are several updates that you must make to the registry when you install or uninstall your performance DLL.

- Creating the application's performance key
- Adding counter names and descriptions to the registry.
- Removing counter names and descriptions from the registry.
- Creating other registry entries

Creating the Application's Performance Key

An application that supports performance counters must have a **Performance** key under the **Services** key. The following example shows the values you should include for this key.

```
HKEY_LOCAL_MACHINE
    \SYSTEM
        \CurrentControlSet
            \Services
                \Application_Name
                    \Performance
                        Library = DLL_Name
                        Open = Open_Function_Name
                        Collect = Collect_Function_Name
                        Close = Close_Function_Name
```

The **Library** value provides the name of the performance DLL, and the **Open**, **Collect**, and **Close** values provide the names of the functions exported from the performance DLL. When a performance monitoring application requests performance data, the system uses these values to determine which performance DLLs to load and which DLL functions to call.

Adding Counter Names and Descriptions to the Registry

The names and descriptions of all performance objects and their counters are stored using the registry. You must add this information for the objects and counters you add to the system.

The following example shows the registry location where performance counter names and descriptions are stored.

```
HKEY_LOCAL_MACHINE
    \SOFTWARE
        \Microsoft
            \Windows NT
                \CurrentVersion
                    \Perflib
                        Last Counter = highest counter index
                        Last Help = highest help index
                        \009
                            Counter = 2 System 4 Memory ...
                            Help = 3 The System object type ...
                        \supported language, other than U.S. English
                            Counter = ...
                            Help = ...
```

To add names and descriptions of the objects and counters for your application, use the **lodctr** utility included with Windows NT. The **lodctr** utility takes strings from an .INI file and adds them to the **Counter** and **Help** values under the appropriate language subkeys. It also updates the **Last Counter** and **Last Help** values. In addition to adding values under the **PerfLib** key, the **lodctr** utility also adds the following value entries to the **Services** node for the application.

```
\HKEY_LOCAL_MACHINE
    \SYSTEM
Error! Bookmark not defined.        \CurrentControlSet
            \Services
                \ApplicationName
                    \Performance
                        First Counter = lowest counter index
                        First Help = lowest help index
                        Last Counter = highest counter index
                        Last Help = highest help index
```

Using lodctr

The command-line syntax for **lodctr** is:

lodctr *MyApplication*.**ini**

Initialization (.INI) File

The .INI file used by **lodctr** has the following format:

```
[info]
applicationname=ApplicationName
symbolfile=SymbolFile

// One key (value optional) for each language supported.
```

```
[languages]
langid=
 .
 .
 .

// Name and description for each counter or counter object
[text]
offset_langid_NAME=Name         // "Counter" name string.
offset_langid_HELP=Description  // "Help" description string.
 .
 .
 .
```

The .INI file entries are variables with the following meanings:

Variable	Description
ApplicationName	The name of the application found under the CurrentControlSet\Services key.
SymbolFile	An .H file containing symbolic offsets of counters. The performance DLL also uses the offsets in this file along with the First Counter and First Help Registry values to determine the indexes of the various counters and counter objects.
Langid	An ID corresponding to the language subkey in the Registry (for example, 009 for U.S. English).
Offset	A symbolic constant defined in *SymbolFile*. Offsets must be consecutive, even numbers beginning with zero. These offsets determine the order in which the counters are installed in the **Counter** and **Help** values in the registry.

The following is an example *SymbolFile*.

```
// SYMFILE.H

#define OBJECT_1    0
#define DEVICE_COUNTER_1    2
#define DEVICE_COUNTER_2    4
```

The following is an example .INI file.

```
// begin .INI file example
[info]
applicationname=ApplicationName
symbolfile=symfile.h

[languages]
009=English
011=OtherLanguage

[text]
OBJECT_1_009_NAME=Device Name
OBJECT_1_009_HELP=Displays performance statistics on Device Name
```

```
OBJECT_1_011_NAME=Device Name in other language
OBJECT_1_011_HELP=Displays performance of Device Name in other language

DEVICE_COUNTER_1_009_NAME=Counter A
DEVICE_COUNTER_1_009_HELP=Displays the current value of Counter A
DEVICE_COUNTER_1_011_NAME=Counter A in other language
DEVICE_COUNTER_1_011_HELP=Displays the value of Counter A in other language

DEVICE_COUNTER_2_009_NAME=Counter B
DEVICE_COUNTER_2_009_HELP=Displays the current rate of Device B
DEVICE_COUNTER_2_011_NAME=Counter B in other language
DEVICE_COUNTER_2_011_HELP=Displays the rate of Device B in other language
```

If you run **lodctr** to add counters for an application and the application does not have a **Services** key, **lodctr** returns without modifying the **Perflib** values.

Note The loading function of LODCTR, **LoadPerfCounterTextStrings**, is declared in LOADPERF.H and exported from LOADPERF.DLL. This allows you to call this function directly from your install program. For example

LoadPerfCounterTextStrings (*MyApplication*.ini, *bQuietModeArg*);

where *MyApplication*.ini is the name of your initialization file and *bQuietModeArg* is a Boolean parameter that indicates whether to display output during the loading of the counter text strings.

Removing Counter Names and Descriptions from the Registry

If you need to remove counter names and descriptions from the registry, use the **unlodctr** utility. This removes the registry entries made by **lodctr**.

Using unlodctr

The command-line syntax for **unlodctr** is:

unlodctr *ApplicationName*

The **unlodctr** utility looks up the **First Counter** and **Last Counter** values in the application's **Performance** key to determine the indexes of the counter objects to remove. Using these indexes, it makes the following changes to the **Perflib** key.

```
HKEY_LOCAL_MACHINE
    \SOFTWARE
        \Microsoft
            \Windows NT
                \CurrentVersion
                    \Perflib
                        Last Counter = updated if changed
                        Last Help = updated if changed
                        \009
                            Counter = application text removed
                            Help    = application text removed
                        \supported language, other than U.S. English
                            Counter = application text removed
                            Help    = application text removed
```

The **unlodctr** utility also removes the **First Counter**, **First Help**, **Last Counter**, and **Last Help** values from the application's **Performance** key.

Note The unloading function of UNLODCTR, **UnloadPerfCounterTextStrings**, is declared in LOADPERF.H and exported from LOADPERF.DLL. This allows you to call this function directly from your uninstall program. For example

UnloadPerfCounterTextStrings (*ApplicationName*, *bQuietModeArg*);

where *ApplicationName* is the name of your application and *bQuietModeArg* is a Boolean parameter that indicates whether to display output during the unloading of the counter text strings.

Creating Other Registry Entries

To obtain the performance data for some applications (those that return counters using the **DeviceIOControl** function), it is necessary to use the **CreateFile** function to open the device associated with the application. In this case, the name specified in **CreateFile** must also be installed in the DOS Devices node of the registry, as shown here:

```
HKEY_LOCAL_MACHINE
    \SYSTEM
        \CurrentControlSet
            \Control
                \Session Manager
                    \DOS Devices
```

Applications that manage multiple device instances with performance data for each device, must put a **Linkage** key in the its **Services** key. The **Linkage** key contains an **Export** value whose data is a list of the device names. For example, a system with two Etherlink cards could have the following registry entries:

```
HKEY_LOCAL_MACHINE
    \SYSTEM
        \CurrentControlSet
            \Services
                \Elnkii
                    \Linkage
                        Export = "\Device\Elnk01" "\Device\Elnk02"
                    \Performance
                        Library = "ElnkStat.dll"
                        Open = "OpenElnkStats"
                        Collect = "GetElnkStats"
                        Close = "CloseElnkStats"
                \Elnk01
                    \Linkage
                    \Parameters
                \Elnk02
                    \Linkage
                    \Parameters
```

When the system calls the Open function in an application's performance DLL, its argument is a string containing the list of device names from the application's Export value, if it is present. The Open function can then use these names to determine the devices for which to collect performance data.

Creating the Performance DLL

Your application's performance DLL defines the counter and object data structures that it uses to pass performance data to the performance monitor application. Your DLL also provides the following exported functions that are called by the system in response to requests for performance data.

Function	Description
Open	Initializes performance monitoring for the application.
Collect	Reports performance data when requested.
Close	Closes performance monitoring.

The prototypes for these functions, and the structures and constants used to define counters and counter objects, are defined in the WINPERF.H file distributed with the Win32 SDK.

Communication between an application and its performance DLL differs for user-mode and privileged-mode applications. The application's performance DLL executes in user mode. Because of this, user-mode applications, such as print and display applications, can use any technique for interprocess communication, such as named pipes, file mapping, or RPC. However, privileged-mode applications must provide an IOCTL interface that returns the performance data to the performance DLL.

How the DLL Interfaces with a Performance Monitor Application

An application retrieves performance data by specifying **HKEY_PERFORMANCE_DATA** in a call to the **RegQueryValueEx** function. If successful, **RegQueryValueEx** fills a buffer of the application with the requested performance data.

The first time an application calls **RegQueryValueEx**, or if the application uses the **RegOpenKey** function to open **HKEY_PERFORMANCE_DATA**, the system calls the Open function for each application with the correct registry entries. This gives each performance DLL an opportunity to initialize its performance data structures. Then, if the Open function returns successfully (or if there is no Open function), the system calls the Collect function. Subsequent calls to **RegQueryValueEx** cause the system to call the Collect function.

When the application has finished collecting performance data, it specifies **HKEY_PERFORMANCE_DATA** in a call to the **RegCloseKey** function. This causes the system to call the Close function for each application. The performance DLLs are then unloaded.

Note It is possible for multiple programs to collect performance data at the same time. The system calls Open and Close functions only once for each application requesting performance data. For remote measurement, the system limits access to these routines to one thread at a time, so synchronization is not a problem. However, for local measurement, because multiple processes may be making simultaneous calls, you must prevent any conflicts from multiple concurrent requests for data.

The Open Function

The system calls the Open function whenever an application first connects to the registry to collect performance data. This function performs the initialization required for the application to provide performance data.

Use the following function prototype for your Open function.

```
DWORD CALLBACK OpenPerformanceData(LPWSTR lpDeviceNames);
```

The name *OpenPerformanceData* is a place-holder for an application-defined name. The *lpDeviceNames* argument points to a buffer containing the Unicode strings stored in the **Export** value in the following registry key:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services*ApplicationName*\Linkage

The strings are REG_MULTI_SZ strings, separated by a UNICODE_NULL, and terminated by two UNICODE_NULL characters. If this entry does not exist, *lpDeviceNames* should be NULL. The strings are the names of the devices managed by this application. The Open function should call the **CreateFile** function to open a handle to each device named. If **CreateFile** fails, the Open function should return the error code returned by the **GetLastError** function; otherwise, it should return ERROR_SUCCESS.

The Open function initializes the data structures it returns to the performance monitor application. In particular, it examines the registry to get the **Counter** and **Help** indices of the objects and counters supported by the application. It then stores these indices in the appropriate members of the **PERF_OBJECT_TYPE** and **PERF_COUNTER_DEFINITION** structures.

Other initialization tasks that might be performed by the Open function include the following:

- Open and map a file mapping object used by the program to store performance data. Create a mutex to prevent the application from changing a value while the performance DLL is reading it. This is one way to communicate performance data from the application to the performance DLL. If you are using another form of interprocess communication, substitute the appropriate steps to initialize that mechanism.
- Initialize event logging, if the DLL uses event logging to report errors. This is the only way for the performance DLL to report error conditions to the user, other than through a debugger. For more information, see Event Logging.
- Initialize the object and counter indexes. This must be done each time the performance DLL is loaded, because the indices vary from machine to machine.
- Perform other initialization tasks necessary for the Collect function to collect performance data efficiently.

The Open function should write any error that prevents the function from completing successfully in the system event log.

The Collect Function

The system calls each application's Collect function whenever a performance monitor program calls the **RegQueryValueEx** function to collect performance data. This function returns the application's performance data.

Use the following function prototype for your Collect function.

```
DWORD WINAPI CollectPerformanceData(
    LPWSTR lpwszValue,
    LPVOID *lppData,
    LPDWORD lpcbBytes,
    LPDWORD lpcObjectTypes);
```

The name CollectPerformanceData is a place-holder for an application-defined name.

Argument	Description
lpwszValue	Points to a string specified by the performance monitor program in a call to the **RegQueryValueEx** function. The string uses one of the formats described in Retrieving Selected Data.
LppData	On input, points to a pointer to the location where the data is to be placed. On successful exit, set **lppData* to the next byte in the buffer available for data, such as one byte past the last byte of your data. The data returned must be a multiple of a **DWORD** in length. It must conform to the **PERF_OBJECT_TYPE** structure, unless this is a collection from a foreign computer. If foreign, any **PERF_OBJECT_TYPE** structures returned must be preceded by a **PERF_DATA_BLOCK** structure for the foreign computer. If the Collect function fails for any reason, leave **lppData* unchanged.
LpcbBytes	On input, points to a 32-bit value that specifies the size, in bytes, of the *lppData* buffer. On successful exit, set **lpcbBytes* to the size, in bytes, of the data written to the *lppData* buffer. This must be a multiple of sizeof(**DWORD**) (a multiple of 4). If the Collect function fails for any reason, set **lpcbBytes* to zero.
lpcObjectTypes	On successful exit, set **lpcObjectTypes* to the number of object type definitions being returned. If the Collect function fails for any reason, it should set **lpcObjectTypes* to zero.

If the requested data specified by *lpwszValue* does not correspond to any of the object indexes or foreign computers supported by your program, leave **lppData* unchanged, and set **lpcbBytes* and **lpcObjectTypes* both to zero. This indicates that no data is returned. If your data collection is time-consuming, you should only respond to specific requests and Costly requests. You should also lower the priority of the thread collecting the data, so that it does not adversely affect system performance.

The Collect function must return one of the values shown in the following table.

Return value	Description
ERROR_MORE_DATA	Indicates that the size of the *lppData* buffer

	as specified by **lpcbBytes* is not large enough to store the data to be returned. In this case, leave **lppData* unchanged, and set **lpcbBytes* and **lpcObjectTypes* to zero. No attempt is made to indicate the required buffer size, because this may change before the next call.
ERROR_SUCCESS	Return this value in all other cases, even if no data is returned or an error occurs. To report errors other than insufficient buffer size, use the system event log, but do not flood the event log with errors on every data collection operation.

To provide more information to the user, the Collect function should write any error that prevents the function from completing successfully in the system event log. For more information, see Event Logging.

If the application collecting the data is running on another machine (remotely), then the extensible counter functions are called in the context of the Winlogon process, which handles the server side of the remote connection. This distinction is important when troubleshooting problems that only occur remotely. The Winlogon process may access the functions using multiple threads, usually one per remote connection. This means that the Open procedure can be called more than once. You should handle this in your code accordingly, so that initialization tasks are not performed more than necessary. The Collect function may also be called concurrently by multiple threads, so you should be careful how you initialize and use temporary data. Static variables should only be used if the data is intended to be shared across threads.

The connection to the machine providing the data should be tested each time data is requested. If the connection cannot be made, you need to return some sort of indication. One suggestion is to provide a Status counter for the object that indicates whether a valid connection exists. When the connection does not exist, set the Status counter and return the last set of valid data.

For foreign computer interfaces, the opening of a channel to the foreign computer must be done in the Collect function because the computer name is not provided to the Open function. The performance DLL should save a handle to the foreign computer to avoid reconnecting on each data collection call, which would significantly slow down system performance.

Once you get the data for a foreign computer, construct a **PERF_DATA_BLOCK** as the first thing in **lppData*, so that your application can return the data. If the system you are measuring does not provide the requested information, provide a reasonable value. You should use values from the local system for time and counter frequency if they are not provided remotely. You may need to use the **PerfTime** and .**PerfFreq** members of the **PERF_OBJECT_TYPE** structure for some counters as well.

Other things that you might do in the Collect include:

- Validate the shared memory pointer and check that the Open function had successfully completed. If the Open function failed, it should have already logged an event, so the Collect function need not report this error.
- Determine the data request type, as described in Retrieving Selected Data. If the request is for Foreign data, it is ignored and the Collect function should return no data. If the request is for a specific object or list of objects, search for its index in the list of object and counter indices. If you do not find a match, the data from this object is not desired, so the function should return no data.
- Request ownership of the mutex for the shared memory object. Wait to get access to the data in the shared memory block.
- Estimate the size of the data to make sure there is enough room for the data in the buffer. If the estimated size is larger than the available size, return a status of ERROR_MORE_DATA. The system passes this error to the thread that issued the call to **RegQueryValueEx** to request the data.

- Copy the performance data from the shared memory to the performance data structure to be returned.
- When all data has been transferred from the shared memory file to the performance data buffer, release the shared memory mutex and update the pointers and counter fields. The updating of the pointers and counter fields is very important, to prevent an access violation from the application, monitor, or system due to misleading buffer length information.

After the collect procedure returns successfully, the system performs the following tests to try and catch logic errors in the collect procedure. The first test to fail will generate an event log message and, in most cases, the data is discarded to prevent any further problems due to invalid pointers. These tests are normally enabled though they can be disabled by changing a registry variable value as described below:

HKEY_LOCAL_MACHINE

\Software

\Microsoft

\Windows NT

\CurrentVersion

\Perflib

ExtCounterTestLevel = *Test_Level*

The *Test_Level* is a REG_DWORD that specifies the test level. Test level 1 requests all tests. Test level 2 requests basic tests. Test level 3 requests no tests. The default is test level 1.

The basic tests performed on the data buffer are as follows:

- Is the *BytesLeft* return value consistent with the returned pointer? The returned value of the *BytesLeft* argument to the Collect procedure is compared to the returned buffer pointer. If all is consistent, the *BytesLeft* value added to the original buffer pointer passed in to the Collect procedure should be the same as the buffer pointer returned by the procedure. If they are not the same a warning message is logged and the *BytesLeft* parameter is replaced by the value computed by finding the difference between the buffer pointer after the function call and the buffer pointer before the function call. This is somewhat risky, in that it assumes the *BytesLeft* value is assumed to be the incorrect one, when in fact it could be that the buffer pointer is the incorrect one.
- Has the returned buffer pointer exceeded the allocated buffer extent? The actual buffer passed to the collect function is allocated specifically for that function call by the performance library and contains a 1K-Byte Guard Page above and below the size indicated by the remaining size of the user's buffer. A separate buffer is used to allow testing of the extensible counter DLL's returned data without corrupting the caller's buffer. If the returned buffer pointer (the pointer to the next byte after this object's data) exceeds the size of this buffer (not including guard pages) then the buffer is assumed to be invalid and discarded since it is too large to be copied into the caller's buffer. This test consists of two parts. If the buffer pointer exceeds the end of the buffer, but not the end of the guard page then a buffer overrun error is logged. If the buffer pointer is past the end of the guard page, then a heap error is logged since the heap that the buffer was allocated from could have been corrupted causing other memory errors.
- Are the guard pages corrupted? The 1K byte Guard Pages above and below the block of memory passed to the collect procedure are initialized with a data pattern before the collect function is called. This data pattern is checked after the collect procedure returns. If any discrepancy is detected a buffer overrun or other memory error is assumed and the buffer is discarded.

The following tests are performed only if test level 1 is used.

- Test object **TotalByteLength** field consistency. This test walks the object(s) returned by the extensible counter to see if the sum of the length of all the objects returned is the same as the value of the size of the returned buffer. Since the Collect function generally returns one or more object structures (including the instance definitions, and counter definitions and data) the sum of each object's length should be the same as the bytes returned. A failure here can indicate the object is not computing the value of the **TotalByteLength** field correctly. This can cause an application program using the data to fail by having it get lost in the data structures.
- **Test Instance ByteLength field consistency**. This test is similar to the test above. The test walks the list of instances in each object that returns multiple instances, to see if the next object or end of buffer follows the last instance. As above, if an inconsistency is detected, the buffer is discarded to prevent the application program from crashing due to a lost pointer.

The Close Function

The system calls each application's Close function when an application calls the **RegCloseKey** function to close the **HKEY_PERFORMANCE_DATA** key. This function performs any cleanup required by the application's performance data collection mechanism. For example, the function could close device handles opened by the **CreateFile** function, or close a handle to a file mapping object.

Use the following function prototype for your Close function.

```
DWORD WINAPI ClosePerformanceData();
```

The name *ClosePerformanceData* is a place-holder for an application-defined name.

The function should return ERROR_SUCCESS.

Error Handling in the DLL

Use event logging to record errors that occur during any of the functions in the performance DLL. Logging error events aids in troubleshooting applications providing performance data during development and after installation. Be careful not to log error events on every Collect call, however, because data collection can be frequent.

The following errors are logged to the event log by the performance library if it has problems with the Open procedure. If one of the errors listed below occurs, then the performance library makes no further calls to this performance DLL. Instead, the DLL is unloaded and data for the object provided by the performance DLL is not returned with the performance data.

- **PERFLIB_OPEN_PROC_NOT_FOUND** - Logged when the procedure name defined in the registry could not be found in the DLL as an exported function. This usually occurs when the performance counter DLL or the service is not installed correctly or the function name has been renamed without updating the installation procedure.
- **PERFLIB_OPEN_PROC_FAILURE** - Logged when the open procedure returned an error status other than ERROR_SUCCESS. This usually occurs when an expected error condition has occurred in the open procedure. Should this happen, the Counter DLL should have also entered an event log entry describing the conditions that caused the failure.
- **PERFLIB_OPEN_PROC_EXCEPTION** - Logged when the open procedure encounters an unhandled exception. This is usually due to an unexpected error condition encountered by the open procedure.

Performance DLL Sample

The Win32 SDK contains a sample performance DLL. This sample is located in the directory MSTOOLS\SAMPLES\WIN32\WINNT\PerfTool.

Performance Data Reference

The following functions and structures are supported for working with performance data.

Performance Data Functions

The following functions are supported for working with performance data.

PDH Interface

CounterPathCallback
PdhAddCounter
PdhBrowseCounters
PdhCalculateCounterFromRawValue
PdhCloseQuery
PdhCollectQueryData
PdhComputeCounterStatistics
PdhConnectMachine
PdhEnumMachines
PdhEnumObjectItems
PdhEnumObjects
PdhExpandCounterPath
PdhGetCounterInfo
PdhGetDefaultPerfCounter
PdhGetDefaultPerfObject
PdhGetFormattedCounterValue
PdhGetRawCounterValue
PdhMakeCounterPath
PdhOpenQuery
PdhParseCounterPath
PdhParseInstanceName
PdhRemoveCounter
PdhSetCounterScaleFactor
PdhValidatePath

PDH Interface (Visual Basic)

PdhAddCounter (VB)
PdhCloseQuery (VB)
PdhCollectQueryData (VB)
PdhCreateCounterPathList (VB)
PdhGetCounterPathElements (VB)
PdhGetCounterPathFromList (VB)
PdhGetDoubleCounterValue (VB)
PdhGetOneCounterPath (VB)
PdhIsGoodStatus (VB)
PdhOpenQuery (VB)
PdhRemoveCounter (VB)

Performance Data Structures

The following structures are supported for working with performance data.

PDH Interface

PDH_BROWSE_DLG_CONFIG
PDH_COUNTER_INFO
PDH_COUNTER_PATH_ELEMENTS
PDH_FMT_COUNTERVALUE
PDH_STATISTICS

Registry Interface

PERF_COUNTER_BLOCK
PERF_COUNTER_DEFINITION
PERF_DATA_BLOCK
PERF_INSTANCE_DEFINITION
PERF_OBJECT_TYPE

PDH Reference

This section describes the PDH data types, functions, and structures.

PDH Data Types

All PDH string structures are shown as "TCHAR", meaning that both Unicode and ANSI string structures are supported. Internally, the native or preferred code path inside the PDH is the Unicode string format, since it is built only for Windows NT. An ANSI version of each function is provided for those functions that have string arguments. They are, however, slightly less efficient since they must convert the ANSI string arguments to Unicode arguments before calling the Unicode version of the function.

The string data types expected are determined by the Unicode symbol when the application is compiled. If Unicode is defined, then the Unicode or wide character strings are expected. If the symbol is undefined, then the 8-bit or ANSI character strings are expected.

Data Type	Description
HQUERY	Handle to a query (HANDLE).
HCOUNTER	Handle to a counter. PDH maintains the linkage between counters and queries.
PDH_STATUS	LONG PDH status value.

Unless otherwise specified, all functions return a Win32 status value of ERROR_SUCCESS if the function completes successfully or a PDH error status value defined in the PDHMSG.H include file.

For data collection and formatting functions it is important to remember that the return value of the function indicates the success or error of the function call and not necessarily that of the counter data. Always check the **CStatus** field of the counter value returned to ensure that the data returned is valid before you use it. If the value of the **CStatus** field does not indicate success, do not use the data.

PDH Functions

This section describes the Performance Data Helper functions.

CounterPathCallback Overview

Group

The **CounterPathCallback** function processes the counter path strings read from the **szReturnPathBuffer** member of the **PDH_BROWSE_DLG_CONFIG** structure.

```
PDH_STATUS __stdcall CounterPathCallback(

  IN DWORD dwArg      // Pointer to a PDH_BROWSE_DLG_CONFIG structure
);
```

Parameters

dwArg

The user-defined **DWORD** value passed to the callback function by the browser. This value is the **dwCallBackArg** member of the **PDH_BROWSE_DLG_CONFIG** structure passed to the browser by the caller.

Remarks

The following members in the **PDH_BROWSE_DLG_CONFIG** structure are used to communicate with the callback function:

szReturnPathBuffer

Contains the counter path strings currently selected by the user.

cchReturnPathLength

Contains the current maximum size of the **szReturnPathBuffer** member. If the callback function reallocates a new buffer, it must also update this value.

CallBackStatus

On entry to the callback function, this member contains the status of the path buffer. On exit, the callback function sets the status value resulting from processing.

If the buffer is too small to load the current selection, the browser will set this value to PDH_MORE_DATA. If the browser sets this member to ERROR_SUCCESS, then the **szReturnPathBuffer** member contains a valid counter path or counter path list.

If the callback function reallocates a new buffer, it should set this member to PDH_RETRY so that the browser will try to load the buffer with the selected paths and call the callback function again.

If some other error occurred, then the callback function should return the appropriate PDH error status value.

Return Values

If the function succeeds (processes the buffer and does not need to be called again), it returns ERROR_SUCCESS.

If the function fails (an error occurs and the function needs to be called again to retry the processing), it returns PDH_RETRY. This is usually the result of insufficient memory.

See Also

PdhBrowseCounters, **PDH_BROWSE_DLG_CONFIG**

PdhAddCounter Overview

Group

The **PdhAddCounter** function initializes a counter structure for the specified counter in the specified query.

```
PDH_STATUS PdhAddCounter(

  IN HQUERY hQuery,                 // handle to the query
  IN LPCTSTR szFullCounterPath,     // path of the counter
  IN DWORD dwUserData,              // user-defined value
  IN HCOUNTER *phCounter            // pointer to the counter handle buffer
);
```

Parameters

hQuery

The handle of the query to which the new counter will be added.

szFullCounterPath

The fully qualified and resolved path of the counter to create. This path cannot contain wild card path strings or characters. See **PDH_COUNTER_PATH_ELEMENTS**.

dwUserData

A user-defined value, typically, a pointer or index to the user's counter structure.

phCounter

A pointer to the buffer that is to receive the handle to the counter that is created.

Return Values

If the function succeeds, it returns ERROR_SUCCESS, creates a new counter, and returns the handle to the counter in the buffer pointed to by *phCounter*.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_CSTATUS_BAD_COUNTERNAME

The counter name path string could not be parsed or interpreted.

PDH_CSTATUS_NO_COUNTER

The specified counter was not found.

PDH_CSTATUS_NO_COUNTERNAME

An empty counter name path string was passed in.

PDH_CSTATUS_NO_MACHINE

A machine entry could not be created.

PDH_CSTATUS_NO_OBJECT

The specified object could not be found.

PDH_FUNCTION_NOT_FOUND

The calculation function for this counter could not be determined.

PDH_INVALID_ARGUMENT

One or more arguments are invalid.

PDH_INVALID_HANDLE

The query handle is not valid.

PDH_MEMORY_ALLOCATION_FAILURE

A memory buffer could not be allocated.

See Also

PdhRemoveCounter, **PDH_COUNTER_PATH_ELEMENTS**

PdhBrowseCounters Overview

Group

The **PdhBrowseCounters** function displays the counter browsing dialog box so that the user can select the counters to be returned to the caller.

```
PDH_STATUS PdhBrowseCounters(

  IN PPDH_BROWSE_DLG_CONFIG pBrowseDlgData      // pointer to dialog structure
);
```

Parameters

pBrowseDlgData

Pointer to a **PDH_BROWSE_DLG_CONFIG** structure that specifies the behavior of the dialog box that receives the counter path strings.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H.

See Also

CounterPathCallback, **PDH_BROWSE_DLG_CONFIG**

PdhCalculateCounterFromRawValue

Overview

Group

The **PdhCalculateCounterFromRawValue** function computes the current value of a counter as referenced by the *hCounter* parameter, using the raw counter data passed in the parameter list.

```
PDH_STATUS PdhCalculateCounterFromRawValue(
  IN HCOUNTER hCounter,                 // handle of the counter
  IN DWORD dwFormat,                    // format code
  IN PPDH_RAW_COUNTER rawValue1,        // first raw counter
  IN PPDH_RAW_COUNTER rawValue2,        // second raw counter
  IN PPDH_FMT_COUNTERVALUE fmtValue     // calculated value
);
```

Parameters

hCounter

The handle of the counter used for the computation. This determines how the data will be computed.

dwFormat

The format code composed of the following values:

Value	Meaning
PDH_FMT_DOUBLE	Return the calculated value as a double-precision floating point real.
PDH_FMT_LARGE	Return the calculated value as a 64-bit integer.
PDH_FMT_LONG	Return the calculated value as a long integer.

The value selected from the previous table can be combined using the **OR** operator with one of the following scaling flags:

Flag	Meaning
PDH_FMT_NOSCALE	Do not apply the counter's scaling factors in the calculation.
PDH_FMT_1000	Multiply the final value by 1000.

rawValue1

A pointer to the raw counter value used to compute the counter value. Some counters (for example, rate counters) require two raw values for a calculation. The format of the buffer depends on the type of counter to be processed.

rawValue2

A pointer to the raw counter value used in computations for counters that require two raw values (for example, rate counters). This value must be the older of the two raw values.

fmtValue

A pointer to the data buffer that will receive the calculated counter value.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

Value	Meaning
PDH_INVALID_ARGUMENT	An argument is not correct or is incorrectly formatted.
PDH_INVALID_HANDLE	The counter handle is not valid.

See Also

PdhGetFormattedCounterValue, PdhGetRawCounterValue, PdhSetCounterScaleFactor

PdhCloseQuery Overview

Group

The **PdhCloseQuery** function closes all counters contained in the specified query, closes all handles related to the query, and frees all memory associated with the query.

```
PDH_STATUS PdhCloseQuery(

  IN HQUERY hQuery      // handle of the query to close and delete.
);
```

Parameters

hQuery

The handle of the query to close and delete.

Return Values

If the function succeeds, it returns ERROR_SUCCESS and closes and deletes the query.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following is a possible error value:

Value	Meaning
PDH_INVALID_HANDLE	The query handle is not valid.

See Also

PdhOpenQuery

PdhCollectQueryData Overview

Group

The **PdhCollectQueryData** function collects the current raw data value for all counters in the specified query and updates the status code of each counter.

```
PDH_STATUS PdhCollectQueryData(

  IN HQUERY hQuery      // handle of the query
);
```

Parameters

hQuery

The handle of the query from which to collect data.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

Value	Meaning
PDH_INVALID_HANDLE	The query handle is not valid.
PDH_NO_DATA	The query does not currently have any counters.

Remarks

The **PdhCollectQueryData** function can succeed, but may not have collected data for all counters. Always check the status code of each counter in the query before using the data.

See Also

PdhOpenQuery

PdhComputeCounterStatistics Overview

Group

The **PdhComputeCounterStatistics** function computes statistics for a counter from an array of raw values.

```
PDH_STATUS PdhComputeCounterStatistics(

  IN HCOUNTER hCounter,                  // handle of the counter
  IN DWORD dwFormat,                     // format code
  IN DWORD dwFirstEntry,                 // first entry in the array
  IN DWORD dwNumEntries,                 // number of entries in the array
  IN PPDH_RAW_COUNTER lpRawValueArray,   // array of raw counter values
  IN PPDH_STATISTICS data                // buffer for counter statistics
);
```

Parameters

hCounter

The handle of the counter used for the computation. This determines how the data will be computed.

dwFormat

The formatting information sent by the caller to indicate how the calculated value should be returned. This parameter can be one of the following values:

PDH_FMT_DOUBLE

Return the calculated value as a double-precision floating point real.

PDH_FMT_LARGE

Return the calculated value as a 64-bit integer.

PDH_FMT_LONG

Return the calculated value as a long integer.

The value selected from the previous table can be combined using the **OR** operator with one of the following scaling flags:

PDH_FMT_NOSCALE

Do not apply the counter's scaling factors in the calculation.

PDH_FMT_1000

Multiply the final value by 1000.

dwFirstEntry

The zero-based index of the first raw counter buffer to look at. This value must point to the oldest entry in the buffer. The **PdhComputeCounterStatistics** function starts at this entry and scans through the buffer, wrapping at the last entry back to the beginning of the buffer and up to the *dwFirstEntry-1* entry, which is assumed to be the newest or most recent data.

dwNumEntries

The number of raw counter entries in the *lpRawValueArray* buffer.

lpRawValueArray

A pointer to the raw counter value or values used to compute the counter value. Some counters (for example, rate counters) require two values for a calculation. The format of the buffer is assumed to be an array of **PDH_RAW_COUNTER** structures that contain *dwNumEntries* entries.

data

A pointer to the **PDH_STATISTICS** buffer to receive the counter statistics.

Return Values

If the function succeeds, it returns ERROR_SUCCESS. The function can return successfully, but the returned data buffer can contain invalid data. Always check the **CStatus** member of the *data* buffer before using the returned statistics.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_INVALID_ARGUMENT

An argument is not correct or is incorrectly formatted.

PDH_INVALID_HANDLE

The counter handle is not valid.

See Also

PdhCalculateCounterFromRawValue, **PdhGetRawCounterValue**, **PDH_RAW_COUNTER**, **PdhSetCounterScaleFactor**, **PDH_STATISTICS**

PdhConnectMachine Overview

Group

The **PdhConnectMachine** function connects to the specified machine, and creates and initializes a machine entry in the PDH DLL.

```
PDH_STATUS PdhConnectMachine(

  IN LPCTSTR szMachineName      // machine to browse
);
```

Parameters

szMachineName

The name of the machine to browse.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_CSTATUS_NO_MACHINE

Unable to connect to the specified machine. Could be caused by the machine not being on, not running Windows NT, not being connected to the network, or having the permissions set on the registry to not allow remote connections or remote performance monitoring by the user.

PDH_MEMORY_ALLOCATION_FAILURE

Unable to allocate a dynamic memory block. Occurs when there is a serious memory shortage in the system due to too many applications running on the system or an insufficient memory paging file.

Remarks

An application can call **PdhConnectMachine** to establish a connection to a remote machine at a more convenient time than when the application opens a query and adds counters.

See Also

PdhEnumMachines

PdhEnumMachines Overview

Group

The **PdhEnumMachines** function returns a list of the names of the machines that have been opened previously by the PDH DLL. The machines listed include those that are currently connected and online, as well as those that are offline or not returning performance data. For information about how to connect to a machine, see **PdhConnectMachine**.

```
PDH_STATUS PdhEnumMachines(

  IN LPCTSR szReserved,              // reserved
  IN LPTSTR mszMachineNameList,      // list of connected machines
  IN LPDWORD pcchBufferLength        // pointer to buffer length
);
```

Parameters

szReserved

Reserved. Must be NULL.

mszMachineNameList

The buffer, allocated by the calling function, to receive the multi-SZ string list of machines to which the PDH DLL is currently connected. This parameter may be NULL if the value in the **DWORD** referenced by *pcchBufferLength* is 0.

pcchBufferLength

A pointer to the **DWORD** containing the size, in characters, of the available buffer on entry and the size of the returned buffer on exit. If the buffer size on entry is zero, then no data is returned in the *mszMachineNameList* buffer (and the pointer may be NULL), and the size of the buffer required, in characters, is returned in the **DWORD** pointed to by *pcchBufferLength*. The size returned includes both terminating NULL characters of the multi-SZ string.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_MORE_DATA

More data is available then the current buffer can hold. Some entries may be returned in the *mszMachinenameList* buffer.

PDH_INSUFFICIENT_BUFFER

The buffer provided is not large enough to contain any data.

PDH_INVALID_ARGUMENT

A required argument is invalid or a reserved argument is not NULL.

See Also

PdhConnectMachine

PdhEnumObjectItems Overview

Group

The **PdhEnumObjectItems** function returns the available counters and instances provided by the specified object on the specified machine.

```
PDH_STATUS PdhEnumObjectItems(

  IN LPCTSTR szReserved,                // reserved
  IN LPCTSTR szMachineName,             // machine name
  IN LPCTSTR szObjectName,              // object name
  IN LPTSTR mszCounterList,             // buffer for object's counters
  IN LPDWORD pcchCounterListLength,     // size of counter list buffer
  IN LPTSTR mszInstanceList,            // buffer for object's instances
  IN LPDWORD pcchInstanceListLength,    // size of instance list buffer
  IN DWORD dwDetailLevel,               // detail level
  IN DWORD dwFlags                      // formatting flag
);
```

Parameters

szReserved

Reserved. Must be NULL.

szMachineName

The name of the machine on which to list the object items. If the machine specified is not in the list of currently connected machines, the PDH will try to connect to it.

szObjectName

The name of the object on the specified machine from which the items are to be listed.

mszCounterList

The buffer allocated by the calling function that is to receive the multi-SZ list of performance counters provided by the specified object on the specified machine. This parameter can be NULL if the value of the **DWORD** referenced by *pcchCounterListLength* is 0.

pcchCounterListLength

A pointer to a **DWORD** containing the size, in characters, of the available buffer, and which returns the size of the buffer used. If the buffer size on entry is zero, then no data is returned in the *mszCounterList* buffer, and the size of the buffer required is returned in the **DWORD** pointed to by *pdwCounterListLength*. The size returned includes both terminating NULL characters of the multi-SZ string.

mszInstanceList

The buffer allocated by the calling function that is to receive the multi-SZ list of the instances of the specified object on the specified machine. This argument can be NULL if the value of the **DWORD** pointed to by *pcchCounterListLength* is 0.

pcchInstanceListLength

A pointer to the **DWORD** containing the size, in characters, of the available buffer, and which returns the size of the buffer used. If the buffer size on entry is zero, then no data is returned in the *mszInstanceList* buffer, and the size of the buffer required is returned in the **DWORD** pointed to by *pcchInstanceListLength*. The size returned includes both terminating NULL characters of the multi-SZ string. If the specified object does not support variable instances, then the returned value will be 0. If

the specified object does support variable instances, but does not currently have any instances, then the value returned in this **DWORD** will be 2, which is the size, in characters, of an empty multi-SZ list string.

dwDetailLevel

The detail level of the performance items to return. All items that are of the specified detail level or less will be returned.

dwFlags

Must be 0.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_MORE_DATA

There are more entries available to return than there is room in the buffer. Some entries might be returned in the buffer, however.

PDH_INSUFFICIENT_BUFFER

The buffer provided is not large enough to contain any data.

PDH_INVALID_ARGUMENT

A required argument is invalid or a reserved argument is not NULL.

PDH_MEMORY_ALLOCATION_FAILURE

A required temporary buffer could not be allocated.

PDH_CSTATUS_NO_MACHINE

The specified machine is offline or unavailable.

PDH_CSTATUS_NO_OBJECT

The specified object could not be found on the specified machine.

See Also

PdhConnectMachine, **PdhEnumObjects**

PdhEnumObjects Overview

Group

The **PdhEnumObjects** function returns a list of objects available on the specified machine.

```
PDH_STATUS PdhEnumObjects(

  IN LPCTSTR szReserved,          // reserved
  IN LPCTSTR szMachineName,       // machine name
  IN LPTSTR mszObjectList,        // buffer for objects
  IN LPDWORD pcchBufferLength,    // size of buffer
  IN DWORD dwDetailLevel,         // detail level
  IN BOOL bRefresh                // refresh flag for connected machines
);
```

Parameters

szReserved

Reserved. Must be NULL.

szMachineName

The name of the machine on which to list the objects. If the machine specified is not in the list of currently connected machines, the PDH will try to connect to the machine and add it to the list.

mszObjectList

The buffer allocated by the calling function that is to receive the multi-SZ list of objects available on the specified machine. This parameter may be NULL if the value of the **DWORD** pointed to by *pcchBufferLength* is 0.

pcchBufferLength

A pointer to the **DWORD** containing the size, in characters, of the available buffer on entry and the size of the returned buffer on exit. If the buffer size on entry is zero, then no data is returned in the *mszObjectList* buffer, and that parameter may be NULL. However, the size of the buffer required, in characters, is returned in the **DWORD** pointed to by *pcchBufferLength*. The size returned includes both terminating NULL characters of the multi-SZ string.

dwDetailLevel

The detail level of the performance items to return. All items that are of the specified detail level or less will be returned.

bRefresh

Indicates that a new list of objects should be obtained from the specified machine. If the machine is not currently connected by the PDH, then this flag is ignored and the list is refreshed automatically.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_MORE_DATA

There are more entries available to return than there is room in the buffer. Some entries might be

returned in the buffer, however.

PDH_INSUFFICIENT_BUFFER

The buffer provided is not large enough to contain any data.

PDH_INVALID_ARGUMENT

A required argument is invalid or a reserved argument is not NULL.

Remarks

When you request a required buffer size, the *bRefresh* flag should be FALSE for all subsequent calls to this function. Otherwise, the size might not be valid.

See Also

PdhConnectMachine, **PdhEnumMachines**

PdhExpandCounterPath Overview

Group

The **PdhExpandCounterPath** function examines the specified machine (or local machine if none is specified) for counters and instances of counters that match the wild card strings in the counter path. The counter path format is assumed to be:

\\machine\object(parent/instance#index)\countername

and the parent, instance, index, and countername elements may contain either a valid name or a wild card character.

```
PDH_STATUS PdhExpandCounterPath(

  IN LPCTSTR szWildCardPath,          // counter path to expand
  IN PPTSTR mszExpandedPathList,      // names that match
  IN LPDWORD pcchPathListLength       // size of buffer
);
```

Parameters

szWildCardPath

The counter path to expand.

mszExpandedPathList

The buffer to receive the list of counter path names that match the wild card specification in the *szWildCardPath* buffer.

pcchPathListLength

The size of the *mszExpandedPathList* buffer, in characters, on entry and the size of the buffer used by the returned path strings on return.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H.

See Also

PdhMakeCounterPath

PdhGetCounterInfo Overview

Group

The **PdhGetCounterInfo** function retrieves information about a counter, such as data size, counter type, path, and user-supplied data values.

```
PDH_STATUS PdhGetCounterInfo(

  IN HQUERY hCounter,                 // handle of the counter
  IN BOOLEAN bRetrieveExplainText,    // TRUE to retrieve explain text
  IN LPDWORD pdwBufferSize,           // pointer to size of lpBuffer
  IN PPDH_COUNTER_INFO lpBuffer       // buffer for counter information
);
```

Parameters

hCounter

The handle of the counter from which to retrieve the information.

bRetrieveExplainText

Determines whether explain text is retrieved. If you set this parameter to TRUE, then the explain text for the counter is retrieved. If you set this parameter to FALSE, the field in the returned buffer is NULL.

pdwBufferSize

A pointer to the size, in bytes, of the buffer passed in *lpBuffer*. If the counter requires a buffer larger than is indicated by *pdwBufferSize*, then **PdhGetCounterInfo** will return the required buffer size in this field. If the function succeeds, then this field will contain the size of the data returned in *lpBuffer*. If the size is zero, then no data will be returned in the buffer (in fact, *lpBuffer* can be NULL); and the size, in bytes, will be returned in the **DWORD**.

lpBuffer

A pointer to the data buffer to receive the counter information. The buffer returned is variable-length, because the string data is appended to the end of the fixed-format portion of the structure. This is done so that all data is returned in a single buffer allocated by the caller.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_INVALID_ARGUMENT

An argument is not correct or is incorrectly formatted.

PDH_INVALID_HANDLE

The counter handle is not valid.

PDH_MORE_DATA

The buffer supplied is not large enough to receive the requested data.

See Also

PDH_COUNTER_INFO

PdhGetDefaultPerfCounter Overview

Group

The **PdhGetDefaultPerfCounter** function retrieves the name of the default counter for the specified object. This can be used to set the initial selection of the counter browser list/combo box.

```
PDH_STATUS PdhGetDefaultPerfCounter(

  IN LPCTSTR szReserved,              // reserved
  IN LPCTSTR szMachineName,           // machine to query
  IN LPCTSTR szObjectName,            // name of object
  IN LPTSTR szDefaultCounterName,     // buffer to receive counter name
  IN LPDWORD pcchBufferSize           // size of counter name
);
```

Parameters

szReserved

Reserved. Must be NULL.

szMachineName

The name of the machine to query.

szObjectName

The name of the object in *szMachineName* to query.

szDefaultCounterName

A pointer to the buffer that will receive the default counter name for the specified object. This parameter can be NULL if the value of the **DWORD** referenced by *pcchBufferSize* is 0.

pcchBufferSize

The size of the *szDefaultCounterName* buffer, in characters, when called, and the size of the counter name returned on return. If the value in the **DWORD** pointed to by this parameter is 0, then the required buffer size, in characters, for this object name will be returned in the **DWORD** and no data will be copied to the buffer.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_INSUFFICIENT_BUFFER

The buffer provided is not large enough for the available data.

PDH_INVALID_ARGUMENT

A required argument is invalid or a reserved argument is not NULL.

PDH_MEMORY_ALLOCATION_FAILURE

A required temporary buffer could not be allocated.

PDH_CSTATUS_NO_MACHINE

The specified machine is offline or unavailable.

PDH_CSTATUS_NO_COUNTERNAME

The default object name cannot be read or found.

PDH_CSTATUS_NO_OBJECT

The specified object could not be found on the specified machine.

PDH_CSTATUS_NO_COUNTER

The default counter was not found in the specified object.

See Also

PdhGetDefaultPerfObject

PdhGetDefaultPerfObject Overview

Group

The **PdhGetDefaultPerfObject** function retrieves the name of the default performance object. The name can be used to select the default entry in the object browser list box.

```
PDH_STATUS PdhGetDefaultPerfObject(

  IN LPCTSTR szReserved,          // reserved
  IN LPCTSTR szMachineName,       // machine to query
  IN LPTSTR szDefaultObjectName,  // buffer to receive object
  IN LPDWORD pcchBufferSize       // size of object name
);
```

Parameters

szReserved

Reserved. Must be NULL.

szMachineName

The name of the machine to query.

szDefaultObjectName

A pointer to the buffer to receive the default object name. This parameter can be NULL if the value of the **DWORD** referenced by *pcchBufferSize* is 0.

pcchBufferSize

The size of the *szDefaultObjectName* buffer, in characters, on entry and the size of the returned object name on exit. If the value in the **DWORD** referenced by this parameter is 0, then the required buffer size, in characters, for this object name will be returned in the **DWORD** and no data will be copied to the buffer.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_INSUFFICIENT_BUFFER

The buffer provided is not large enough for the available data.

PDH_INVALID_ARGUMENT

A required argument is invalid or a reserved argument is not NULL.

PDH_MEMORY_ALLOCATION_FAILURE

A required temporary buffer could not be allocated.

PDH_CSTATUS_NO_MACHINE

The specified machine is offline or unavailable.

PDH_CSTATUS_NO_COUNTERNAME

The default object name cannot be read or found.

See Also

PdhGetDefaultPerfCounter

PdhGetFormattedCounterValue Overview

Group

The **PdhGetFormattedCounterValue** function returns the current value of a specified counter in the format requested by the caller.

```
PDH_STATUS PdhGetFormattedCounterValue(

  IN HCOUNTER hCounter,                 // handle of the counter
  IN DWORD dwFormat,                    // formatting flag
  IN LPDWORD lpdwType,                  // counter type
  IN PPDH_FMT_COUNTERVALUE pValue       // counter value
);
```

Parameters

hCounter

The handle of the counter whose current value is to be formatted and returned.

dwFormat

The formatting information sent by the caller to indicate how the data should be returned. This parameter can be one of the following values:

PDH_FMT_DOUBLE

Return data as a double-precision floating point real.

PDH_FMT_LARGE

Return data as a 64-bit integer.

PDH_FMT_LONG

Return data as a long integer.

The value selected from the previous table can be combined using the **OR** operator with one of the following scaling flags:

PDH_FMT_NOSCALE

Do not apply the default scaling factor.

PDH_FMT_1000

Multiply the actual value by 1000.

lpdwType

A pointer to a **DWORD** buffer that will receive the counter type. The possible counter types are described in WINPERF.H. This parameter is optional.

pValue

A pointer to the data buffer that will receive the counter value.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_INVALID_ARGUMENT

An argument is not correct or is incorrectly formatted.

PDH_INVALID_DATA

The specified counter does not contain valid data or a successful status code.

PDH_INVALID_HANDLE

The counter handle is not valid.

Remarks

The data for the counter is locked (protected) for the duration of the call to **PdhGetFormattedCounterValue** to prevent any changes during the processing of the call. Reading the data (calling this function successfully) clears the data-changed flag for the counter.

See Also

PdhGetRawCounterValue, **PdhSetCounterScaleFactor**

PdhGetRawCounterValue Overview

Group

The **PdhGetRawCounterValue** function returns the current raw value of the counter.

```
PDH_STATUS PdhGetRawCounterValue(

  IN HCOUNTER hCounter,              // handle of the counter
  IN LPDWORD lpdwType,               // counter type
  IN PPDH_RAW_COUNTER pValue         // counter value
);
```

Parameters

hCounter

The handle of the counter from which to retrieve the current raw value.

lpdwType

A pointer to a **DWORD** buffer that will receive the counter type. The possible counter types are described in WINPERF.H. This parameter is optional. You can pass NULL if you don't want the type information.

pValue

A pointer to the data buffer that will receive the counter value.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_INVALID_ARGUMENT

An argument is not correct or is incorrectly formatted.

PDH_INVALID_HANDLE

The counter handle is not valid.

Remarks

The data for the counter is locked (protected) for the duration of the call to **PdhGetRawCounterValue** to prevent any changes during processing of the call.

See Also

PdhGetFormattedCounterValue, PdhCalculateCounterFromRawValue

PdhMakeCounterPath Overview

Group

The **PdhMakeCounterPath** function creates a full counter path using the members specified in the structure passed in the parameter list.

```
PDH_STATUS PdhMakeCounterPath(

  IN PDH_COUNTER_PATH_ELEMENTS *pCounterPathElements,   // counter path elements
  IN LPTSTR szFullPathBuffer,                           // path string buffer
  IN LPDWORD pcchBufferSize,                            // size of buffer
  IN DWORD dwFlags                                      // reserved
);
```

Parameters

**pCounterPathElements*

Pointer to a **PDH_COUNTER_PATH_ELEMENTS** structure that contains the individual members that are to make up the path. The following table indicates which members are required and which are optional:

Member	Required or Optional
LPTSTR szMachineName	optional
LPTSTR szObjectName	required
LPTSTR szInstanceName	optional
LPTSTR szParentInstance	optional
DWORD dwInstanceIndex	optional ((**DWORD**)-1 if no index)
LPTSTR szCounterName	required

If a required member is not present, then no path string will be constructed. If the **szMachineName** member is NULL, then no machine name will be included in the path (a generic path will be created). If the instance name argument is NULL, then no instance reference will be inserted into the path string and the **szParentInstance** and the **dwInstanceIndex** members will be ignored.

szFullPathBuffer

The buffer allocated by the caller that will receive the full path string created by this function.

pcchBufferSize

A pointer to the **DWORD** containing the size, in bytes, of the available buffer, and which returns with the size of the buffer used. If the buffer size on entry is zero, then no data is returned in the *szFullPathBuffer* buffer, however, the size of the buffer required is returned in the **DWORD**. The size returned includes the terminating NULL character of the string.

dwFlags

Reserved. Must be zero.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H.

See Also

PDH_COUNTER_PATH_ELEMENTS, PdhParseCounterPath

PdhOpenQuery Overview

Group

The **PdhOpenQuery** function creates and initializes a unique query structure that is used to manage collection of performance data.

```
PDH_STATUS PdhOpenQuery(

  IN LPVOID pReserved,      // reserved
  IN DWORD dwUserData,      // a value associated with this query
  IN HQUERY *phQuery        // pointer to a buffer that will receive the query handle
);
```

Parameters

pReserved

Reserved. Must be NULL.

dwUserData

A user-defined **DWORD** value that is to be associated with this query. You can call **PdhGetCounterInfo** to retrieve the user data value for the query in which a counter resides.

phQuery

A pointer to the buffer to receive the handle to the query that is created.

Return Values

If the function succeeds, it returns ERROR_SUCCESS, creates a new query, and returns the handle to the query in the buffer pointed to by *phQuery*.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_INVALID_ARGUMENT

One or more arguments are invalid.

PDH_MEMORY_ALLOCATION_FAILURE

A memory buffer could not be allocated.

See Also

PdhCloseQuery, **PdhGetCounterInfo**

PdhParseCounterPath Overview

Group

The **PdhParseCounterPath** function parses the elements of the counter path and stores the results in the structure passed by the caller.

```
PDH_STATUS PdhParseCounterPath(

  IN LPCTSTR szFullPathBuffer,                          // path string buffer
  IN PDH_COUNTER_PATH_ELEMENTS *pCounterPathElements,   // counter path elements
  IN LPDWORD pdwBufferSize,                             // size of buffer
  IN DWORD dwFlags                                      // reserved
);
```

Parameters

szFullPathBuffer

The buffer containing the counter path to parse into individual components.

pCounterPathElements

A pointer to a **PDH_COUNTER_PATH_ELEMENTS** structure that will receive the individual components of the path referenced by the *szFullPathBuffer* parameter. The buffer space allocated for this should be large enough for the structure and the strings that will be referenced by the members in this structure. As this function parses the elements in the path, they are stored in the buffer after the **PDH_COUNTER_PATH_ELEMENTS** structure. This allows the calling function to allocate, and eventually free, only a single block of memory for this structure and the strings referenced by the members of this structure, rather than having the calling function allocate the structure and the string buffers separately.

pdwBufferSize

A pointer to the **DWORD** containing the size, in bytes, of the buffer referenced by *pCounterPathElements* and returns with the size of the buffer used. If the buffer size on entry is zero, then no data is returned in the *pCounterPathElements* buffer, however, the size of the buffer required is returned in the **DWORD** referenced by this pointer.

dwFlags

Reserved. Must be zero.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_INVALID_ARGUMENT

An argument is incorrect, or this function does not have the necessary access to that argument.

PDH_INSUFFICIENT_BUFFER

The buffer supplied is not large enough to accept the resulting data.

PDH_INVALID_PATH

The path is not formatted correctly and cannot be parsed.

PDH_MEMORY_ALLOCATION_FAILURE

A temporary buffer cannot be allocated.

See Also

PDH_COUNTER_PATH_ELEMENTS, **PdhMakeCounterPath**

PdhParseInstanceName Overview

Group

The **PdhParseInstanceName** function parses the elements of an instance string and returns them in the buffers supplied by the caller.

```
PDH_STATUS PdhParseInstanceName(

  IN LPCTSTR szInstanceString,          // instance string
  IN LPTSTR szInstanceName,             // parsed instance name
  IN LPDWORD pcchInstanceNameLength,    // length of name
  IN LPTSTR szParentName,               // name of parent index
  IN LPDWORD pcchParentNameLength,      // length of parent name
  IN LPDWORD lpIndex                    // instance index
);
```

Parameters

szInstanceString

A pointer to the string containing the instance substring to parse into individual components. This string can contain the following formats, and is less than MAX_PATH characters in length:

instance
instance#index
parent/instance
parent/instance#index

szInstanceName

A pointer to the buffer that will receive the instance name parsed from the instance string. This pointer can be NULL if the **DWORD** referenced by the *pcchInstanceNameLength* parameter is 0.

pcchInstanceNameLength

A pointer to the **DWORD** that contains the length of the *szInstanceName* buffer. If the value of this **DWORD** is 0, then the buffer size required to hold the instance name will be returned.

szParentName

A pointer to the buffer that will receive the name of the parent index if one is specified. This parameter can be NULL if the value of the **DWORD** referenced by the *pcchParentNameLength* parameter is 0.

pcchParentNameLength

A pointer to the **DWORD** that contains the length of the *szParentName* buffer. If the value of this **DWORD** is 0, then the buffer size required to hold the instance name will be returned.

lpIndex

A pointer to the **DWORD** that will receive the index value of the instance. If an index entry is not present in the string, then this value will be 0. This argument can be NULL if this information is not needed.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_INVALID_ARGUMENT

An argument is invalid or incorrect.

PDH_INSUFFICIENT_BUFFER

One or both of the string buffers supplied is not large enough for the strings to be returned.

PDH_INVALID_INSTANCE

The instance string is incorrectly formatted, exceeds MAX_PATH characters in length, or cannot be parsed.

See Also

PdhMakeCounterPath

PdhRemoveCounter Overview

Group

The **PdhRemoveCounter** function removes a counter from a query.

```
PDH_STATUS PdhRemoveCounter(

  IN HCOUNTER hCounter      // handle of the counter
);
```

Parameters

hCounter

The handle of the counter to remove from its query.

Return Values

If the function succeeds, it returns ERROR_SUCCESS and removes the specified counter from its query.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following is a possible error value:

PDH_INVALID_HANDLE

The counter handle is not valid.

Remarks

After you remove a counter, any references to it using the handle passed in *hCounter* will be invalid and will return an error.

See Also

PdhAddCounter, PdhOpenQuery

PdhSetCounterScaleFactor Overview

Group

The **PdhSetCounterScaleFactor** function sets the scale factor that is applied to the calculated value of the specified counter when you request the formatted counter value. If the PDH_FMT_NOSCALE flag is set, then this scale factor is ignored.

```
PDH_STATUS PdhSetCounterScaleFactor(

  IN HCOUNTER hCounter,     // handle of the counter
  IN LONG lFactor           // power of ten used to multiply value
);
```

Parameters

hCounter

The handle of the counter to receive the scale factor in *lFactor*.

lFactor

The power of ten by which to multiply the calculated value before returning it. The valid range of this parameter is PDH_MIN_SCALE (-7) (the returned value is the actual value times 10-7) to PDH_MAX_SCALE (+7) (the returned value is the actual value times 10+7). A value of zero will set the scale to one, so that the actual value is returned.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is an error status defined in PDHMSG.H. The following are possible error values:

PDH_INVALID_ARGUMENT

The scale value is out of range.

PDH_INVALID_HANDLE

The counter handle is not valid.

See Also

PdhCalculateCounterFromRawValue, **PdhComputeCounterStatistics**, **PdhGetFormattedCounterValue**

PdhValidatePath Overview

Group

The **PdhValidatePath** function validates that the specified counter is present on the machine specified in the counter path.

```
PDH_STATUS PdhValidatePath(

  IN LPCTSTR szFullCounterPath       // counter path to validate
);
```

Parameters

szFullCounterPath

The counter path to validate.

Return Values

If the function succeeds, it returns ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHMSG.H. The following are possible error values:

PDH_CSTATUS_NO_INSTANCE

The specified instance of the performance object was not found.

PDH_CSTATUS_NO_COUNTER

The specified counter was not found in the performance object.

PDH_CSTATUS_NO_OBJECT

The specified performance object was not found on the machine.

PDH_CSTATUS_NO_MACHINE

The specified machine could not be found or connected to.

PDH_CSTATUS_BAD_COUNTERNAME

The counter path string could not be parsed.

PDH_MEMORY_ALLOCATION_FAILURE

The function is unable to allocate a required temporary buffer.

See Also

PdhMakeCounterPath

PDH Functions for Visual Basic 4.0

The functions described in this section enable the Visual Basic programmer to use Windows NT Performance Data Helper within a Visual Basic application program.

PdhAddCounter (VB)

The Visual Basic **PdhAddCounter** function creates a counter entry in the specified query object, and returns the handle for that counter upon successful completion.

PdhAddCounter(

ByVal *QueryHandle* as **Long,**
ByVal *CounterPath* as **String,**
ByRef *CounterHandle* as **Long)**
as **Long**

Parameters

QueryHandle

The ID of the query to assign this counter to. This must be a value returned by a successful call to **PdhOpenQuery**.

CounterPath

The text string containing the name of the counter path to add to the query. The contents of this string must be a valid counter path, as obtained from the counter browser or other source.

CounterHandle

The unique reference that identifies this counter in the query. This variable must be initialized to zero before the function is called. It contains a valid value on return only if the function completes successfully.

Return Values

If the function succeeds, it returns a **Long** integer equal to ERROR_SUCCESS and a new handle in the *CounterHandle* variable.

If the function fails, the return value is a PDH error status defined in PDHDEFS.TXT. The following are possible error values:

PDH_INVALID_ARGUMENT

One or more of the arguments is invalid or incorrect.

PDH_MEMORY_ALLOCATION_FAILURE

A memory buffer could not be allocated.

PDH_INVALID_HANDLE

The query handle is not valid.

PDH_CSTATUS_NO_COUNTER

The specified counter was not found.

PDH_CSTATUS_NO_OBJECT

The specified object could not be found.

PDH_CSTATUS_NO_MACHINE

A machine entry could not be created.

PDH_CSTATUS_BAD_COUNTERNAME

An empty counter name path string was passed in.

PDH_FUNCTION_NOT_FOUND

The calculation function for this counter could not be determined.

PdhCloseQuery (VB)

The Visual Basic **PdhCloseQuery** function closes the specified query and all counters related to that query. All resources allocated for the query and its counters are freed. The handles of counters added to this query should be deleted or zeroed and not used after the query they belong to is deleted.

PdhCloseQuery(

ByVal *QueryHandle* as **Long)**
as **Long**

Parameters

QueryHandle

The ID of the query to close and delete. Closing a query closes all the counters associated with the query.

Return Values

If the function succeeds, it returns a **Long** integer equal to ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHDEFS.TXT. The following is a possible error value:

PDH_INVALID_HANDLE

The handle is invalid.

PdhCollectQueryData (VB)

The Visual Basic **PdhCollectQueryData** function collects the current raw value of each counter in the query referenced by the *QueryHandle* parameter and updates the internal buffers of each counter object.

PdhCollectQueryData(

ByVal *QueryHandle* as **Long)**
as **Long**

Parameters

QueryHandle

The ID of the query to update. This must be a value returned by a call to **PdhOpenQuery**.

Return Values

If the function succeeds, it returns a **Long** integer equal to ERROR_SUCCESS. Note that the function can return successfully, but some or all counters in the query may not have been updated.

If the function fails, the return value is a PDH error status defined in PDHDEFS.TXT. The following are possible error values:

PDH_INVALID_HANDLE

The query handle is not valid.

PDH_NO_DATA

The query does not have any counters defined yet.

PdhCreateCounterPathList (VB)

The Visual Basic **PdhCreateCounterPathList** function displays the performance counter browsing dialog box, which lets the user select several performance counters. Each selected counter path must then be read using the **PdhGetCounterPathFromList** function.

PdhCreateCounterPathList(

ByVal *DetailLevel* as **Long,**
ByVal *CaptionString* as **String)**
as **Long**

Parameters

DetailLevel

One of the following values defined in PDHDEFS.TXT. The selected value indicates the types of counters to be displayed in the dialog box:

Value	Counters Displayed
PERF_DETAIL_NOVICE	Only counters that the novice user is likely to understand.
PERF_DETAIL_ADVANCED	Counters that the advanced user is likely to understand, in addition to the novice-user counters.
PERF_DETAIL_EXPERT	Counters that the expert user and software developer is likely to understand, in addition to the counters for the novice and advanced users.
PERF_DETAIL_WIZARD	All counters in the system.

CaptionString

A string variable that contains the text that will be displayed in the caption bar of the dialog box.

Return Values

The function returns the number of counter paths that the user selected.

PdhGetCounterPathElements (VB)

The Visual Basic **PdhGetCounterPathElements** function parses a fully qualified performance counter path string into its individual elements. Each of the string variables must be the same size (*BufferSize*) and dimensioned and initialized before it is used in this function.

PdhGetCounterPathElements(

ByVal *PathString* as **String,**
ByVal *MachineName* as **String,**
ByVal *ObjectName* as **String,**
ByVal *InstanceName* as **String,**
ByVal *ParentInstance* as **String,**
ByVal *CounterName* as **String,**
ByVal *BufferSize* as **Long)**
as **Long**

Parameters

PathString

The counter path string that is to be broken up into its individual elements.

MachineName

The string to receive the Machine name element of the counter path specified in *PathString*.

ObjectName

The string to receive the Object element of the counter path specified in *PathString*.

InstanceName

The string to receive the Instance name element, if used, of the counter path specified in *PathString*.

ParentInstance

The string to receive the Parent Instance element, if used, of the counter path specified in *PathString*.

CounterName

The string to receive the Machine name element of the counter path specified in *PathString*.

BufferSize

The maximum size of each string variable used as a parameter to this function call.

Return Values

If the function succeeds, it returns a **Long** integer equal to ERROR_SUCCESS.

If the function fails, the return value is a PDH error status defined in PDHDEFS.TXT. The following are possible error values:

PDH_INVALID_ARGUMENT

One or more of the string buffers is not the correct size.

PDH_INSUFFICIENT_BUFFER

One or more of the counter path elements is too large for the return buffer length.

PDH_MEMORY_ALLOCATION_FAILURE

A temporary memory buffer could not be allocated.

PdhGetCounterPathFromList (VB)

The Visual Basic **PdhGetCounterPathFromList** function copies the counter path referenced by the *Index* parameter from a counter path list created by the user from the most recent call to the **PdhCreateCounterPathList** function.

```
PdhGetCounterPathFromList(

  ByVal Index as Long,
    ByVal Buffer as String,
  ByVal BufferLength as Long)
      as Long
```

Parameters

Index

The index of the counter path to retrieve. This must be a value that is greater than or equal to 1, and less than or equal to the value returned by the **PdhCreateCounterPathList** function.

Buffer

A dimensioned and initialized string that will receive the counter path that corresponds to the value of the *Index* parameter.

BufferLength

The maximum number of characters that will fit in the string referenced by *Buffer*.

Return Values

The function returns the number of characters copied to *Buffer*.

PdhGetDoubleCounterValue (VB)

The Visual Basic **PdhGetDoubleCounterValue** function returns the current value of the specified counter as a double-precision floating point value. You should check *CounterStatus* before using the returned number, because the counter may not be valid when it is read. Call **PdhIsGoodStatus** to check the *CounterStatus*.

PdhGetDoubleCounterValue(

ByVal *CounterHandle* as **Long,**
ByRef *CounterStatus* as **Long)**
as **Double**

Parameters

CounterHandle

The ID of the counter whose current value is to be read.

CounterStatus

The variable in which the current status of the counter value is returned to the caller. The returned data value is valid if and only if the value returned in *CounterStatus* is PDH_CSTATUS_VALID_DATA or PDH_CSTATUS_NEW_DATA (defined in PDHDEFS.TXT). If the value returned in *CounterStatus* is any other value, do not use the data.

Return Values

PdhGetDoubleCounterValue returns the double-precision floating point value of the current counter, computed and formatted as defined by the counter type.

PdhGetOneCounterPath (VB)

The Visual Basic **PdhGetOneCounterPath** function displays a dialog box that lets the user browse the available performance counters on a Windows NT system and select one counter. The counter selected is returned in the *PathString* variable. The *PathString* variable must be dimensioned and initialized before this function is called, and the dimensioned size must be indicated by the *PathLength* variable.

PdhGetOneCounterPath(

ByVal *PathString* as **String,**
ByVal *PathLength* as **Long,**
ByVal *DetailLevel* as **Long,**
ByVal *CaptionString* as **String)**
as **Long**

Parameters

PathString

The initialized string variable used to receive the counter path selected by the user.

PathLength

The length of the initialized *PathString*.

DetailLevel

One of the following values defined in PDHDEFS.TXT. The selected value indicates the types of counters to be displayed in the dialog box:

Value	Counters Displayed
PERF_DETAIL_NOVICE	Only counters that the novice user is likely to understand.
PERF_DETAIL_ADVANCED	Counters that the advanced user is likely to understand, in addition to the novice-user counters.
PERF_DETAIL_EXPERT	Counters that the expert user and software developer is likely to understand, in addition to the counters for the novice and advanced users.
PERF_DETAIL_WIZARD	All counters in the system.

CaptionString

A string variable that contains the text that will be displayed in the caption bar of the dialog box.

Return Values

The function returns the number of characters written to the *PathString* buffer.

PdhIsGoodStatus (VB)

The Visual Basic **PdhIsGoodStatus** function tests a status value to determine if it is a success or failure code. If the status value is a successful one, then the return value will be nonzero. If it is a failure status code, the return value will be zero.

PdhIsGoodStatus(

ByVal *StatusValue* as **Long)**
as **Long**

Parameters

StatusValue

The PDH status value returned by another PDH function that is to be tested.

Return Values

The function returns zero if the status code is a failure status code. It returns nonzero if the status code is a success status code.

PdhOpenQuery (VB)

The Visual Basic **PdhOpenQuery** function creates and initializes a unique query structure that is used to manage the collection of performance data.

PdhOpenQuery(

ByRef *QueryHandle* as **Long)**
as **Long**

Parameters

QueryHandle

A variable that is cleared (equals 0) before the function is called and, if the function is successful, contains the unique ID of the query that is created and opened. This handle is used in the subsequent calls to other PDH functions to identify the query.

Return Values

If the function succeeds, it returns a **Long** integer equal to ERROR_SUCCESS and a new handle in the *QueryHandle* variable.

If the function fails, the return value is a PDH error status defined in PDHDEFS.TXT. The following are possible error values:

PDH_INVALID_ARGUMENT

The argument is invalid or incorrect.

PDH_MEMORY_ALLOCATION_FAILURE

A temporary memory buffer could not be allocated.

PdhRemoveCounter (VB)

The Visual Basic **PdhRemoveCounter** function removes the counter entry identified by the *CounterHandle* parameter.

```
PdhRemoveCounter(

  ByVal CounterHandle as Long)
    as Long
```

Parameters

CounterHandle

The ID of the counter returned by the **PdhAddCounter** function.

Return Values

If the function succeeds, it returns a **Long** integer equal to ERROR_SUCCESS and the counter entry is deleted from the query.

If the function fails, the return value is a PDH error status defined in PDHDEFS.TXT. The following is a possible error value:

Value	Meaning
PDH_INVALID_HANDLE	The counter handle is not valid.

PDH Structures

The following are Performance Data Helper structures.

PDH_RAW_COUNTER

The **PDH_RAW_COUNTER** structure returns the data as it was collected from the counter provider. No translation, formatting, or other interpretation is performed on the data.

```
typedef struct _PDH_RAW_COUNTER {
    DWORD       CStatus;
    FILETIME    TimeStamp;
    LONGLONG    FirstValue;
    LONGLONG    SecondValue;
    DWORD       MultiCount;
} PDH_RAW_COUNTER, *PPDH_RAW_COUNTER;
```

Members

CStatus

The status of the last collection operation for this counter. See Counter Status versus Function Status.

TimeStamp

The local time this data was collected.

FirstValue

The first raw counter value.

SecondValue

The second (if necessary) raw counter value.

MultiCount

The multi count (normally 1).

See Also

PdhCalculateCounterFromRawValue, PdhComputeCounterStatistics, PdhGetRawCounterValue

PDH_FMT_COUNTERVALUE Overview

Group

The **PDH_FMT_COUNTERVALUE** structure is the most commonly used method of reading the data from within an application. The data type of the data returned (member of the union that will have meaningful data) is specified by the caller when requesting the data. The **CStatus** member provides the status of the counter. Check this member before using the data in a calculation or display. The counter, and therefore the data, could be invalid.

```
typedef struct _PDH_FMT_COUNTERVALUE {
    DWORD    CStatus;
    union {
        LONG        longValue;
        double      doubleValue;
        LONGLONG    largeValue;
    };
} PDH_FMT_COUNTERVALUE, *PPDH_FMT_COUNTERVALUE;
```

Members

CStatus

The status of last collection operation for this counter. See Counter Status versus Function Status.

See Also

PdhCalculateCounterFromRawValue, PdhGetFormattedCounterValue

PDH_STATISTICS Overview

Group

The **PDH_STATISTICS** structure is used to return the statistics of the values in an array of raw counters managed by the application program. The format of the data in the **PDH_FMT_COUNTERVALUE** structure is described in the **dwFormat** member.

```
typedef struct _PDH_STATISTICS {
    DWORD                   dwFormat;
    DWORD                   count;
    PDH_FMT_COUNTERVALUE    min;
    PDH_FMT_COUNTERVALUE    max;
    PDH_FMT_COUNTERVALUE    mean;
} PDH_STATISTICS, *PPDH_STATISTICS;
```

Members

dwFormat

The format of the data.

count

The number of values in the array.

min

The minimum of the values.

max

The maximum of the values.

mean

The mean of the values.

See Also

PdhComputeCounterStatistics, **PDH_FMT_COUNTERVALUE**

PDH_COUNTER_PATH_ELEMENTS Overview

Group

The **PDH_COUNTER_PATH_ELEMENTS** structure contains the parsed elements of a fully qualified counter path. The main purpose of this structure is to provide formatted text to the calling application for display.

```
typedef struct _PDH_COUNTER_PATH_ELEMENTS {
    LPTSTR  szMachineName;
    LPTSTR  szObjectName;
    LPTSTR  szInstanceName;
    LPTSTR  szParentInstance;
    DWORD   dwInstanceIndex;
    LPTSTR  szCounterName;
} PDH_COUNTER_PATH_ELEMENTS, *PPDH_COUNTER_PATH_ELEMENTS;
```

Members

szMachineName

The machine name, parsed from the counter path.

szObjectName

The object name, parsed from the counter path.

szInstanceName

The instance name, parsed from counter path.

szParentInstance

The parent instance name, parsed from counter path.

dwInstanceIndex

The index of duplicate instance names.

szCounterName

The counter name.

See Also

PdhMakeCounterPath, PdhParseCounterPath

PDH_COUNTER_INFO Overview

Group

The **PDH_COUNTER_INFO** structure contains information describing the properties of a counter. This information also includes the counter path. The format of this buffer is to have the structure described below followed by a variable-length buffer containing the string information referenced by the string pointers in the structure (**szFullPath**, **szMachineName**, **szObjectName**, **szInstanceName**, **szParentInstance**, **szCounterName**, and **szExplainText**).

```
typedef struct _PDH_COUNTER_INFO {
    DWORD   dwLength;
    DWORD   dwType;
    DWORD   CVersion;
    DWORD   CStatus;
    LONG    lScale;
    LONG    lDefaultScale;
    DWORD   dwUserData;
    DWORD   dwQueryUserData;
    LPTSTR  szFullPath;
    union   {
        PDH_COUNTER_PATH_ELEMENTS CounterPath;
        struct {
            LPTSTR   szMachineName;
            LPTSTR   szObjectName;
            LPTSTR   szInstanceName;
            LPTSTR   szParentInstance;
            DWORD    dwInstanceIndex;
            LPTSTR   szCounterName;
        };
    };
    LPTSTR  szExplainText;
    DWORD   DataBuffer[1];
} PDH_COUNTER_INFO, *PPDH_COUNTER_INFO;
```

Members

dwLength

The length of the structure, including strings.

dwType

The counter type.

CVersion

Counter version information.

CStatus

The current counter status. See Counter Status versus Function Status.

lScale

The current scale factor.

lDefaultScale

The recommended scale factor.

dwUserData

The value of the counter's **User Data** field.

dwQueryUserData

The value of the **User Data** field for the query to which the counter belongs.

szFullPath

The full counter path.

CounterPath

See **PDH_COUNTER_PATH_ELEMENTS**.

szExplainText

The explain text for this counter.

DataBuffer[1]

The first byte of string data appended to the structure.

See Also

PDH_COUNTER_PATH_ELEMENTS, **PdhGetCounterInfo**

PDH_BROWSE_DLG_CONFIG Overview

Group

The **PDH_BROWSE_DLG_CONFIG** structure is used by the **PdhBrowseCounters** function to configure the **PDH Browse Counters** dialog box.

```
typedef struct _BrowseDlgConfig {
    // Configuration flags
    DWORD   bIncludeInstanceIndex:1,
            bSingleCounterPerAdd:1,
            bSingleCounterPerDialog:1,
            bLocalCountersOnly:1,
            bWildCardInstances:1,
            bHideDetailBox:1,
            bInitializePath:1,
            bReserved:25;

    HWND                hWndOwner;
    LPTSTR              szReserved;
    LPTSTR              szReturnPathBuffer;
    DWORD               cchReturnPathLength;
    CounterPathCallBack pCallBack;
    DWORD               dwCallBackArg;
    PDH_STATUS          CallBackStatus;
    DWORD               dwDefaultDetailLevel;
    LPTSTR              szDialogBoxCaption;
} PDH_BROWSE_DLG_CONFIG, *PPDH_BROWSE_DLG_CONFIG;
```

Members

Configuration Flags

The initial **DWORD** of the **PDH_BROWSE_DLG_CONFIG** structure contains the following configuration flag fields. The default value for each field is FALSE.

bIncludeInstanceIndex

TRUE: The returned counter path will include the instance index for the instance.

FALSE: The returned counter path will not contain an instance number.

bSingleCounterPerAdd

TRUE: Multiple selections are not permitted. Only one counter will be returned each time the **ADD** button is clicked.

FALSE: Multiple and wild card selections are permitted. Selected counters are returned as a multi-SZ string each time the **ADD** button is clicked.

bSingleCounterPerDialog

TRUE: The dialog box is closed after the **ADD** button is clicked the first time.

FALSE: The dialog box is not closed until the **CLOSE** button is clicked. The **ADD** button can be clicked, and counters can be added, multiple times.

bLocalCountersOnly

TRUE: The machine name will not prefix the counter path in the returned string.

FALSE: The machine name will prefix the counter path unless the user selects the **Local Counters Only** radio button.

bWildCardInstances

TRUE: If wild card instances (for example, ALL) are selected, then the returned counter path will include the wild-card syntax (for example, "*") for the instance field.

FALSE: If **All Instances** has been selected, all the instances currently found for that object will be returned in a multi-SZ string.

bHideDetailBox

TRUE: Removes the **Detail Level** combo box from the dialog box so the user cannot change the detail level of the counters displayed in the dialog box. The detail level will be fixed to the value of the **dwDefaultDetailLevel** member.

FALSE: Displays the **Detail Level** combo box in the dialog box, allowing the user to change the detail level of the counters displayed.

Note that the counters displayed will be those whose detail level is less than or equal to the current detail level selection. Selecting a detail level of **Wizard** will display all counters and objects.

bInitializePath

TRUE: Selects the counter and object based on the first counter path contained in the **szReturnPathBuffer** member when the dialog box is first displayed, instead of using the default counter and object specified by the machine.

FALSE: Selects the initial counter and object using the default counter and object information returned by the machine.

hWndOwner

The handle of the calling function's window.

szReserved

Reserved. Must be NULL.

szReturnPathBuffer

The buffer, allocated by the caller, that is used to initialize the first selection in the list boxes (**bInitializePath == TRUE)** and to return the selected counters to the calling function or the callback procedure.

cchReturnPathLength

The current maximum size, in characters, of the buffer referenced by the **szReturnPathBuffer** member.

pCallBack

The address of the callback function used to update application buffers and controls when a multiple-selection dialog box is configured.

dwCallBackArg

User-defined argument that is passed as the only argument to the callback function when it is called by the dialog box.

CallBackStatus

Status of the dialog box prior to calling the callback function.

dwDefaultDetailLevel

The default detail level to show on startup in the **Detail Level** combo box. If the **Detail Level** combo box is not shown, this is the detail level to use in filtering the displayed performance counters and objects.

szDialogBoxCaption

The optional caption to be displayed in the caption bar of the **PDH Browse Counters** dialog box. If this member is NULL, the caption will be **Browse Performance Counters**.

See Also

CounterPathCallback, **PdhBrowseCounters**

PERF_COUNTER_BLOCK Quick Info

Overview

Group

The **PERF_COUNTER_BLOCK** structure contains the length, in bytes, of the performance-counter data. This structure is followed by data for the number of counters specified in the **PERF_OBJECT_TYPE** structure.

```
typedef struct _PERF_COUNTER_BLOCK { // pcd
    DWORD ByteLength;
} PERF_COUNTER_BLOCK;
```

Members

ByteLength

Specifies the length, in bytes, of this structure, including the counters that follow.

Remarks

This structure is part of the performance data provided by the **RegQueryValueEx** function when the **HKEY_PERFORMANCE_DATA** key is used.

See Also

PERF_OBJECT_TYPE, **RegQueryValueEx**

PERF_COUNTER_DEFINITION Quick Info

The **PERF_COUNTER_DEFINITION** structure describes a performance counter. The Unicode names in this structure must appear in a message file.

```
typedef struct _PERF_COUNTER_DEFINITION { // pcd
    DWORD  ByteLength;
    DWORD  CounterNameTitleIndex;
    LPWSTR CounterNameTitle;
    DWORD  CounterHelpTitleIndex;
    LPWSTR CounterHelpTitle;
    DWORD  DefaultScale;
    DWORD  DetailLevel;
    DWORD  CounterType;
    DWORD  CounterSize;
    DWORD  CounterOffset;
} PERF_COUNTER_DEFINITION;
```

Members

ByteLength

Contains the length, in bytes, of this structure.

CounterNameTitleIndex

Contains the index of the counter name in the title database of the registry.

CounterNameTitle

Points to the name of the counter. This member contains NULL, initially, but it can contain a pointer to the actual string once the string is located.

CounterHelpTitleIndex

Contains the index to the counter's Help title in the title database of the registry.

CounterHelpTitle

Points to the title of Help. This member contains NULL, initially, but it can contain a pointer to the actual string once the string is located.

DefaultScale

Specifies the power of 10 by which to scale a chart line, assuming the vertical axis is 100. If this value is zero, the scale value is 1; if this value is 1, the scale value is 10; if this value is -1, the scale value is .10; and so on.

DetailLevel

Specifies the level of detail for the counter. Applications use this value to control display complexity. This member can be one of the following values:

Value	Meaning
PERF_DETAIL_NOVICE	The data can be understood by the uninformed user.
PERF_DETAIL_ADVANCED	The data is designed for the advanced user.
PERF_DETAIL_EXPERT	The data is designed for the expert

	user.
PERF_DETAIL_WIZARD	The data is designed for the system designer.

CounterType

Specifies the type of counter. This member is some combination of the following values.

These values indicate the counter's data size:

Value	Meaning
PERF_SIZE_DWORD	The counter data is a doubleword.
PERF_SIZE_LARGE	The counter data is a large integer.
PERF_SIZE_ZERO	The counter data is a zero-length field.
PERF_SIZE_VARIABLE_LEN	The size of the counter data is in the **CounterSize** member.

These values indicate the additional contents of this member:

Value	Meaning
PERF_TYPE_NUMBER	The counter data is a number value but not a counter.
PERF_TYPE_COUNTER	The counter data is an increasing numeric value.
PERF_TYPE_TEXT	The counter data is a text field.
PERF_TYPE_ZERO	The counter data is always zero.

If PERF_TYPE_NUMBER is specified, one of these values is also specified to indicate the format of the number:

Value	Meaning
PERF_NUMBER_HEX	The counter data should be displayed as a hexadecimal value.
PERF_NUMBER_DECIMAL	The counter data should be displayed as a decimal value.
PERF_NUMBER_DEC_1000	The counter data should be divided by 1000 and displayed as a decimal value.

If PERF_TYPE_COUNTER is specified, one of these values is also specified to indicate the type of counter:

Value	Meaning
PERF_COUNTER_VALUE	The counter value is valid without additional calculation; that is, it should be displayed as is.

PERF_COUNTER_RATE	The counter value should be divided by the elapsed time.
PERF_COUNTER_FRACTION	The counter value should be divided by the base value indicated by the next counter if it is of type PERF_COUNTER_BASE or by the value of the counter subtype.
PERF_COUNTER_BASE	The counter value is the base value to use in fractions.
PERF_COUNTER_ELAPSED	The counter value is a start time to be subtracted from the current time.
PERF_COUNTER_QUEUELEN	The performance application should use the **Queuelen** counter – that is, the Queue Length Space-Time Product formula. The next counter is the number currently in the queue. Multiply it by the current time (units specified by this counter's subtype). Add the product to the original value of the counter. To obtain the average queue length, divide the result of the addition by the delta time.
PERF_COUNTER_HISTOGRAM	The counter value begins or ends a histogram.

If PERF_TYPE_COUNTER is specified, one of these values is also specified to indicate the subtype of counter:

Value	**Meaning**
PERF_TIMER_TICK	The frequency of the high-resolution performance counter should be used as the base.
PERF_TIMER_100NS	The time base units of the 100-nanosecond timer should be used as the base.
PERF_OBJECT_TIMER	The object-timer frequency should be used as the base unit. This value is system-defined in this counter's PERF_OBJECT_TYPE definition.

If PERF_TYPE_TEXT is specified, one of these values is also specified to indicate the type of text:

Value	**Meaning**
PERF_TEXT_UNICODE	The counter data contains Unicode text.
PERF_TEXT_ASCII	The counter data contains ASCII text.

These values indicate how to use the counter data in a calculation:

Value	Meaning
PERF_DELTA_COUNTER	The difference between the previous counter value and the current counter value is computed before proceeding.
PERF_DELTA_BASE	The difference between the previous base value and the current base value is computed before proceeding.
PERF_INVERSE_COUNTER	After other calculations, the counter should be inverted before displaying or converting to a percentage.
PERF_MULTI_COUNTER	This value is a sum of counters from several sources, the number of which is indicated by the next counter.

These values indicate the display suffix of the counter data:

Value	Meaning
PERF_DISPLAY_NO_SUFFIX	There is no display suffix.
PERF_DISPLAY_PER_SEC	The display suffix is '/sec'.
PERF_DISPLAY_PERCENT	The display suffix is '%'.
PERF_DISPLAY_SECONDS	The display suffix is 'secs'.
PERF_DISPLAY_NOSHOW	The counter value should not be displayed.

CounterSize

Specifies the counter size, in bytes.

CounterOffset

Specifies the offset from the start of the **PERF_COUNTER_BLOCK** structure to the first byte of this counter.

Remarks

This structure is part of the performance data provided by the **RegQueryValueEx** function when the **HKEY_PERFORMANCE_DATA** key is used.

See Also

RegQueryValueEx

PERF_DATA_BLOCK Quick Info

Group

Group

The **PERF_DATA_BLOCK** structure describes the performance data provided by **RegQueryValueEx** function. The data starts with a **PERF_DATA_BLOCK** structure and is followed by a **PERF_OBJECT_TYPE** structure and other object-specific data for each type of object monitored.

```
typedef struct _PERF_DATA_BLOCK { // pdb
    WCHAR         Signature[4];
    DWORD         LittleEndian;
    DWORD         Version;
    DWORD         Revision;
    DWORD         TotalByteLength;
    DWORD         HeaderLength;
    DWORD         NumObjectTypes;
    DWORD         DefaultObject;
    SYSTEMTIME    SystemTime;
    LARGE_INTEGER PerfTime;
    LARGE_INTEGER PerfFreq;
    LARGE_INTEGER PerfTime100nSec;
    DWORD         SystemNameLength;
    DWORD         SystemNameOffset;
} PERF_DATA_BLOCK;
```

Members

Signature

Contains the Unicode string PERF.

LittleEndian

Contains zero if the processor is big endian and one if it is little endian.

Version

Contains the version of the performance (**PERF_**) structures. This member is greater than or equal to one.

Revision

Contains the revision of the performance (**PERF_**) structures. This member is greater than or equal to zero.

TotalByteLength

Contains the total length, in bytes, of the performance data.

HeaderLength

Contains the length, in bytes, of this structure.

NumObjectTypes

Contains the number of object types being monitored.

DefaultObject

Contains the object title index of the default object whose performance data is to be displayed. This member can be -1 to indicate that no data is to be displayed.

SystemTime

Contains the time when the system is monitored. This member is in Coordinated Universal Time (UTC) format.

PerfTime

Contains the performance-counter value, in counts, for the system being monitored.

PerfFreq

Contains the performance-counter frequency, in counts per second, for the system being monitored.

PerfTime100nSec

Contains the performance-counter value, in 100 nanosecond units, for the system being monitored.

SystemNameLength

Contains the length, in bytes, of the system name.

SystemNameOffset

Contains the offset from the beginning of this structure to the name of the system being monitored.

See Also

PERF_OBJECT_TYPE, **RegQueryValueEx**

PERF_INSTANCE_DEFINITION Quick Info

Overview

Overview

The **PERF_INSTANCE_DEFINITION** structure contains the instance-specific information for a block of performance data. There is one **PERF_INSTANCE_DEFINITION** structure for each instance specified in the PERF_OBJECT_TYPE structure.

```
typedef struct _PERF_INSTANCE_DEFINITION { // pid
    DWORD ByteLength;
    DWORD ParentObjectTitleIndex;
    DWORD ParentObjectInstance;
    DWORD UniqueID;
    DWORD NameOffset;
    DWORD NameLength;
} PERF_INSTANCE_DEFINITION;
```

Members

ByteLength

Specifies the length, in bytes, of this structure, including the subsequent name.

ParentObjectTitleIndex

Specifies the index of the name of the "parent" object in the title database. For example, if the object is a thread, the parent object type is a process, or if the object is a logical drive, the parent is a physical drive.

ParentObjectInstance

Specifies the index to an instance of the parent object type that is the parent of this instance. This member may be zero or greater.

UniqueID

Specifies the unique identifier used instead of the instance name. This member is PERF_NO_UNIQUE_ID if there is no such identifier.

NameOffset

Specifies the offset from the beginning of this structure to the Unicode name of this instance.

NameLength

Specifies the length, in bytes, of the instance name. This member is zero if the instance does not have a name.

See Also

PERF_OBJECT_TYPE

PERF_OBJECT_TYPE Quick Info

Overview

Overview

The **PERF_OBJECT_TYPE** structure describes object-specific performance information. This structure is followed by a list of **PERF_COUNTER_DEFINITION** structures, one for each counter defined for the type of object.

```
typedef struct _PERF_OBJECT_TYPE {  // pot
    DWORD  TotalByteLength;
    DWORD  DefinitionLength;
    DWORD  HeaderLength;
    DWORD  ObjectNameTitleIndex;
    LPWSTR ObjectNameTitle;
    DWORD  ObjectHelpTitleIndex;
    LPWSTR ObjectHelpTitle;
    DWORD  DetailLevel;
    DWORD  NumCounters;
    DWORD  DefaultCounter;
    DWORD  NumInstances;
    DWORD  CodePage;
    LARGE_INTEGER PerfTime;
    LARGE_INTEGER PerfFreq;
} PERF_OBJECT_TYPE;
```

Members

TotalByteLength

Contains the length, in bytes, of the object-specific data. This value includes this structure, the **PERF_COUNTER_DEFINITION** structures, and the **PERF_INSTANCE_DEFINITION** and **PERF_COUNTER_BLOCK** structures for each instance. This member specifies the offset from the beginning of this structure to the next **PERF_OBJECT_TYPE** structure if one exists.

DefinitionLength

Contains the length, in bytes, of the object-specific data. This value includes this structure and the **PERF_COUNTER_DEFINITION** structures for this object. This member is the offset from the beginning of the **PERF_OBJECT_TYPE** structure to the first **PERF_INSTANCE_DEFINITION** structure or to the **PERF_COUNTER_DEFINITION** structures if there is no instance data.

HeaderLength

Contains the length, in bytes, of this structure. This member is the offset to the first **PERF_COUNTER_DEFINITION** structure for this object.

ObjectNameTitleIndex

Contains the index to the object's name in the title database.

ObjectNameTitle

Points to the name of the object. This member initially contains NULL, but it can contain a pointer to the actual string once the string is located.

ObjectHelpTitleIndex

Contains the index to the object's Help title in the title database.

ObjectHelpTitle

Points to the title of Help. This member initially contains NULL, but it can contain a pointer to the actual string once the string is located.

DetailLevel

Specifies the level of detail. Applications use this value to control display complexity. This value is the minimum detail level of all the counters for a given object. This member can be one of the following values:

Detail level	Meaning
PERF_DETAIL_NOVICE	No technical ability is required to understand the counter data.
PERF_DETAIL_ADVANCED	The counter data is provided for advanced users.
PERF_DETAIL_EXPERT	The counter data is provided for expert users.
PERF_DETAIL_WIZARD	The counter data is provided for system designers.

NumCounters

Specifies the number of counters in each counter block. There is one counter block per instance.

DefaultCounter

Specifies the default counter whose information is to be displayed when this object is selected. This member is typically greater than or equal to zero. However, this member may be -1 to indicate that there is no default.

NumInstances

Specifies the number of object instances for which counters are being provided.

CodePage

Specifies the code page. This member is zero if the instance strings are in Unicode. Otherwise, this member is the code-page identifier of the instance names.

PerfTime

Specifies the current value, in counts, of the high-resolution performance counter.

PerfFreq

Specifies the current frequency, in counts per second, of the high-resolution performance counter.

Remarks

If there is only one instance of the object type, the counter definitions are followed by a single **PERF_COUNTER_BLOCK** structure. This structure is followed by data for each counter. (The **PERF_COUNTER_BLOCK** structure contains the total length of the structure and the counter data that follows it.)

If there is more than one instance of the object type, the list of counter definitions is followed by a **PERF_INSTANCE_DEFINITION** structure and a **PERF_COUNTER_BLOCK** structure for each instance. The **PERF_INSTANCE_DEFINITION** structure includes the name, the identifier, and the name of the parent of the instance.

Following the counter data, there is a **PERF_INSTANCE_DEFINITION** structure and a **PERF_COUNTER_BLOCK** structure for each instance specified in the **PERF_DATA_BLOCK** structure

that begins the performance-data area.

See Also

PERF_COUNTER_BLOCK, PERF_COUNTER_DEFINITION, PERF_INSTANCE_DEFINITION