Chapter 1: System Information and Control

The system services described in this chapter operate on the system as a whole rather than on individual objects within the system. They mostly gather information about the performance and operation of the system and set system parameters.

ZwQuerySystemInformation

ZwQuerySystemInformation queries information about the system.

```
NTSYSAPI
NTSTATUS
NTAPI
ZwQuerySystemInformation(
  IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
  IN OUT PVOID SystemInformation,
  IN ULONG SystemInformationLength,
  OUT PULONG ReturnLength OPTIONAL
  );
```

Parameters

SystemInformationClass

The type of system information to be queried. The permitted values are a subset of the enumeration `SYSTEM_INFORMATION_CLASS`, described in the following section.

SystemInformation

Points to a caller-allocated buffer or variable that receives the requested system information.

SystemInformationLength

The size in bytes of `SystemInformation`, which the caller should set according to the given `SystemInformationClass`.

ReturnLength

Optionally points to a variable that receives the number of bytes actually returned to `SystemInformation`; if `SystemInformationLength` is too small to contain the available information, the variable is normally set to zero except for two information classes (6 and 11) when it is set to the number of bytes required for the available information. If this information is not needed, `ReturnLength` may be a null pointer.

Return Value

Returns `STATUS_SUCCESS` or an error status, such as `STATUS_INVALID_INFO_CLASS`, `STATUS_NOT_IMPLEMENTED`, or `STATUS_INFO_LENGTH_MISMATCH`.

Related Win32 Functions

`GetSystemInfo`, `GetTimeZoneInformation`, `GetSystemTimeAdjustment`, PSAPI functions, and performance counters.

Remarks

ZwQuerySystemInformation is the source of much of the information displayed by "Performance Monitor" for the classes Cache, Memory, Objects, Paging File, Process, Processor, System, and Thread. It is also frequently used by resource kit utilities that display information about the system.

The `ReturnLength` information is not always valid (depending on the information class), even when the routine returns `STATUS_SUCCESS`. When the return value indicates `STATUS_INFO_LENGTH_MISMATCH`, only some of the information classes return an estimate of the required length.

Some information classes are implemented only in the "checked" version of the kernel. Some, such as `SystemCallCounts`, return useful information only in "checked" versions of the kernel.

Some information classes require certain flags to have been set in `NtGlobalFlags` at boot time. For example, `SystemObjectInformation` requires that `FLG_MAINTAIN_OBJECT_TYPELIST` be set at boot time.

Information class `SystemNotImplemented1 (4)` would return `STATUS_NOT_IMPLEMENTED` if it were not for the fact that it uses `DbgPrint` to print the text "`EX: SystemPathInformation now available via SharedUserData.`" and then calls `DbgBreakPoint`. The breakpoint exception is caught by a frame based exception handler (in the absence of intervention by a debugger) and causes **ZwQuerySystemInformation** to return with `STATUS_BREAKPOINT`.

ZwSetSystemInformation

ZwSetSystemInformation sets information that affects the operation of the system.

```
NTSYSAPI
NTSTATUS
NTAPI
ZwSetSystemInformation(
  IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
  IN OUT PVOID SystemInformation,
  IN ULONG SystemInformationLength
  );
```

Parameters

SystemInformationClass

> The type of system information to be set. The permitted values are a subset of the enumeration `SYSTEM_INFORMATION_CLASS`, described in the following section.

SystemInformation

Points to a caller-allocated buffer or variable that contains the system information to be set.

SystemInformationLength

The size in bytes of `SystemInformation`, which the caller should set according to the given `SystemInformationClass`.

Return Value

Returns `STATUS_SUCCESS` or an error status, such as `STATUS_INVALID_INFO_CLASS`, `STATUS_NOT_IMPLEMENTED` or `STATUS_INFO_LENGTH_MISMATCH`.

Related Win32 Functions

`SetSystemTimeAdjustment`.

Remarks

At least one of the information classes uses the `SystemInformation` parameter for both input and output.

SYSTEM_INFORMATION_CLASS

The system information classes available in the "free" (retail) build of the system are listed below along with a remark as to whether the information class can be queried, set, or both. Some of the information classes labeled "`SystemNotImplementedXxx`" are implemented in the "checked" build, and a few of these classes are briefly described later.

```
                                        Query  Set
typedef enum _SYSTEM_INFORMATION_CLASS {
  SystemBasicInformation,                // 0    Y    N
  SystemProcessorInformation,            // 1    Y    N
  SystemPerformanceInformation,          // 2    Y    N
  SystemTimeOfDayInformation,            // 3    Y    N
  SystemNotImplemented1,                 // 4    Y    N
  SystemProcessesAndThreadsInformation,  // 5    Y    N
  SystemCallCounts,                      // 6    Y    N
  SystemConfigurationInformation,        // 7    Y    N
  SystemProcessorTimes,                  // 8    Y    N
  SystemGlobalFlag,                      // 9    Y    Y
  SystemNotImplemented2,                // 10    Y    N
  SystemModuleInformation,              // 11    Y    N
  SystemLockInformation,                // 12    Y    N
  SystemNotImplemented3,                // 13    Y    N
  SystemNotImplemented4,                // 14    Y    N
  SystemNotImplemented5,                // 15    Y    N
  SystemHandleInformation,              // 16    Y    N
  SystemObjectInformation,              // 17    Y    N
  SystemPagefileInformation,            // 18    Y    N
```

```
  SystemInstructionEmulationCounts,     // 19    Y    N
  SystemInvalidInfoClass1,              // 20
  SystemCacheInformation,               // 21    Y    Y
  SystemPoolTagInformation,             // 22    Y    N
  SystemProcessorStatistics,            // 23    Y    N
  SystemDpcInformation,                 // 24    Y    Y
  SystemNotImplemented6,                // 25    Y    N
  SystemLoadImage,                      // 26    N    Y
  SystemUnloadImage,                    // 27    N    Y
  SystemTimeAdjustment,                 // 28    Y    Y
  SystemNotImplemented7,                // 29    Y    N
  SystemNotImplemented8,                // 30    Y    N
  SystemNotImplemented9,                // 31    Y    N
  SystemCrashDumpInformation,           // 32    Y    N
  SystemExceptionInformation,           // 33    Y    N
  SystemCrashDumpStateInformation,      // 34    Y    Y/N
  SystemKernelDebuggerInformation,      // 35    Y    N
  SystemContextSwitchInformation,       // 36    Y    N
  SystemRegistryQuotaInformation,       // 37    Y    Y
  SystemLoadAndCallImage,               // 38    N    Y
  SystemPrioritySeparation,             // 39    N    Y
  SystemNotImplemented10,               // 40    Y    N
  SystemNotImplemented11,               // 41    Y    N
  SystemInvalidInfoClass2,              // 42
  SystemInvalidInfoClass3,              // 43
  SystemTimeZoneInformation,            // 44    Y    N
  SystemLookasideInformation,           // 45    Y    N
  SystemSetTimeSlipEvent,               // 46    N    Y
  SystemCreateSession,                  // 47    N    Y
  SystemDeleteSession,                  // 48    N    Y
  SystemInvalidInfoClass4,              // 49
  SystemRangeStartInformation,          // 50    Y    N
  SystemVerifierInformation,            // 51    Y    Y
  SystemAddVerifier,                    // 52    N    Y
  SystemSessionProcessesInformation     // 53    Y    N
} SYSTEM_INFORMATION_CLASS;
```

SystemBasicInformation

```
typedef struct _SYSTEM_BASIC_INFORMATION { // Information Class 0
  ULONG Unknown;
  ULONG MaximumIncrement;
  ULONG PhysicalPageSize;
  ULONG NumberOfPhysicalPages;
  ULONG LowestPhysicalPage;
  ULONG HighestPhysicalPage;
  ULONG AllocationGranularity;
  ULONG LowestUserAddress;
  ULONG HighestUserAddress;
  ULONG ActiveProcessors;
  UCHAR NumberProcessors;
} SYSTEM_BASIC_INFORMATION, *PSYSTEM_BASIC_INFORMATION;
```

Members

Unknown

Always contains zero; interpretation unknown.

MaximumIncrement

The maximum number of 100-nanosecond units between clock ticks. Also the number of 100-nanosecond units per clock tick for kernel intervals measured in clock ticks.

PhysicalPageSize

The size in bytes of a physical page.

NumberOfPhysicalPages

The number of physical pages managed by the operating system.

LowestPhysicalPage

The number of the lowest physical page managed by the operating system (numbered from zero).

HighestPhysicalPage

The number of the highest physical page managed by the operating system (numbered from zero).

AllocationGranularity

The granularity to which the base address of virtual memory reservations is rounded.

LowestUserAddress

The lowest virtual address potentially available to user mode applications.

HighestUserAddress

The highest virtual address potentially available to user mode applications.

ActiveProcessors

A bit mask representing the set of active processors in the system. Bit 0 is processor 0; bit 31 is processor 31.

NumberProcessors

The number of processors in the system.

Remarks

Much of the data in this information class can be obtained by calling the Win32 function `GetSystemInfo`.

SystemProcessorInformation

```
typedef struct _SYSTEM_PROCESSOR_INFORMATION { // Information Class 1
  USHORT ProcessorArchitecture;
  USHORT ProcessorLevel;
  USHORT ProcessorRevision;
  USHORT Unknown;
  ULONG FeatureBits;
} SYSTEM_PROCESSOR_INFORMATION, *PSYSTEM_PROCESSOR_INFORMATION;
```

Members

ProcessorArchitecture

The system's processor architecture. Some of the possible values are defined in winnt.h with identifiers of the form `PROCESSOR_ARCHITECTURE_*` (where '*' is a wildcard).

ProcessorLevel

The system's architecture-dependent processor level. Some of the possible values are defined in the Win32 documentation for the `SYSTEM_INFO` structure.

ProcessorRevision

The system's architecture-dependent processor revision. Some of the possible values are defined in the Win32 documentation for the `SYSTEM_INFO` structure.

Unknown

Always contains zero; interpretation unknown.

FeatureBits

A bit mask representing any special features of the system's processor (for example, whether the Intel MMX instruction set is available). The flags for the Intel platform include:

Intel Mnemonic	Value	Description
VME	0x0001	Virtual-8086 Mode Enhancements
TCS	0x0002	Time Stamp Counter
	0x0004	CR4 Register
CMOV	0x0008	Conditional Mov/Cmp Instruction
PGE	0x0010	PTE Global Bit
PSE	0x0020	Page Size Extensions
MTRR	0x0040	Memory Type Range Registers
CXS	0x0080	CMPXCHGB8 Instruction
MMX	0x0100	MMX Technology
PAT	0x0400	Page Attribute Table
FXSR	0x0800	Fast Floating Point Save and Restore
SIMD	0x2000	Streaming SIMD Extension

Remarks

Much of the data in this information class can be obtained by calling the Win32 function `GetSystemInfo`.

SystemPerformanceInformation

```
typedef struct _SYSTEM_PERFORMANCE_INFORMATION { // Information Class 2
  LARGE_INTEGER IdleTime;
  LARGE_INTEGER ReadTransferCount;
  LARGE_INTEGER WriteTransferCount;
  LARGE_INTEGER OtherTransferCount;
  ULONG ReadOperationCount;
  ULONG WriteOperationCount;
  ULONG OtherOperationCount;
  ULONG AvailablePages;
  ULONG TotalCommittedPages;
  ULONG TotalCommitLimit;
  ULONG PeakCommitment;
  ULONG PageFaults;
  ULONG WriteCopyFaults;
  ULONG TransitionFaults;
  ULONG Reserved1;
  ULONG DemandZeroFaults;
  ULONG PagesRead;
  ULONG PageReadIos;
  ULONG Reserved2[2];
  ULONG PagefilePagesWritten;
  ULONG PagefilePageWriteIos;
  ULONG MappedFilePagesWritten;
  ULONG MappedFilePageWriteIos;
  ULONG PagedPoolUsage;
  ULONG NonPagedPoolUsage;
  ULONG PagedPoolAllocs;
  ULONG PagedPoolFrees;
  ULONG NonPagedPoolAllocs;
  ULONG NonPagedPoolFrees;
  ULONG TotalFreeSystemPtes;
  ULONG SystemCodePage;
  ULONG TotalSystemDriverPages;
  ULONG TotalSystemCodePages;
  ULONG SmallNonPagedLookasideListAllocateHits;
  ULONG SmallPagedLookasideListAllocateHits;
  ULONG Reserved3;
  ULONG MmSystemCachePage;
  ULONG PagedPoolPage;
  ULONG SystemDriverPage;
  ULONG FastReadNoWait;
  ULONG FastReadWait;
  ULONG FastReadResourceMiss;
  ULONG FastReadNotPossible;
  ULONG FastMdlReadNoWait;
  ULONG FastMdlReadWait;
  ULONG FastMdlReadResourceMiss;
  ULONG FastMdlReadNotPossible;
  ULONG MapDataNoWait;
  ULONG MapDataWait;
  ULONG MapDataNoWaitMiss;
  ULONG MapDataWaitMiss;
  ULONG PinMappedDataCount;
```

```
  ULONG PinReadNoWait;
  ULONG PinReadWait;
  ULONG PinReadNoWaitMiss;
  ULONG PinReadWaitMiss;
  ULONG CopyReadNoWait;
  ULONG CopyReadWait;
  ULONG CopyReadNoWaitMiss;
  ULONG CopyReadWaitMiss;
  ULONG MdlReadNoWait;
  ULONG MdlReadWait;
  ULONG MdlReadNoWaitMiss;
  ULONG MdlReadWaitMiss;
  ULONG ReadAheadIos;
  ULONG LazyWriteIos;
  ULONG LazyWritePages;
  ULONG DataFlushes;
  ULONG DataPages;
  ULONG ContextSwitches;
  ULONG FirstLevelTbFills;
  ULONG SecondLevelTbFills;
  ULONG SystemCalls;
} SYSTEM_PERFORMANCE_INFORMATION, *PSYSTEM_PERFORMANCE_INFORMATION;
```

Members

IdleTime

The total idle time, measured in units of 100-nanoseconds, of all the processors in the system.

ReadTransferCount

The number of bytes read by all calls to **ZwReadFile**.

WriteTransferCount

The number of bytes written by all calls to **ZwWriteFile**.

OtherTransferCount

The number of bytes transferred to satisfy all other I/O operations, such as **ZwDeviceIoControlFile.**

ReadOperationCount

The number of calls to **ZwReadFile**.

WriteOperationCount

The number of calls to **ZwWriteFile**.

OtherOperationCount

The number of calls to all other I/O system services, such as **ZwDeviceIoControlFile**.

AvailablePages

The number of pages of physical memory available to processes running on the system.

TotalCommittedPages

The number of pages of committed virtual memory.

TotalCommitLimit

The number of pages of virtual memory that could be committed without extending the system's pagefiles.

PeakCommitment

The peak number of pages of committed virtual memory.

PageFaults

The number of page faults (both soft and hard).

WriteCopyFaults

The number of page faults arising from attempts to write to copy-on-write pages.

TransitionFaults

The number of soft page faults (excluding demand zero faults).

DemandZeroFaults

The number of demand zero faults.

PagesRead

The number of pages read from disk to resolve page faults.

PageReadIos

The number of read operations initiated to resolve page faults.

PagefilePagesWritten

The number of pages written to the system's pagefiles.

PagefilePageWriteIos

The number of write operations performed on the system’s pagefiles.

MappedFilePagesWritten

The number of pages written to mapped files.

MappedFilePageWriteIos

The number of write operations performed on mapped files.

PagedPoolUsage

The number of pages of virtual memory used by the paged pool.

NonPagedPoolUsage

The number of pages of virtual memory used by the nonpaged pool.

PagedPoolAllocs

The number of allocations made from the paged pool.

PagedPoolFrees

The number of allocations returned to the paged pool.

NonPagedPoolAllocs

The number of allocations made from the nonpaged pool.

NonPagedPoolFrees

The number of allocations returned to the nonpaged pool.

TotalFreeSystemPtes

The number of available System Page Table Entries.

SystemCodePage

The number of pages of pageable operating system code and static data in physical memory. The meaning of "operating system code and static data" is defined by address range (lowest system address to start of system cache) and includes a contribution from win32k.sys.

TotalSystemDriverPages

The number of pages of pageable device driver code and static data.

TotalSystemCodePages

The number of pages of pageable operating system code and static data. The meaning of "operating system code and static data" is defined by load time (`SERVICE_BOOT_START` driver or earlier) and does not include a contribution from win32k.sys.

SmallNonPagedLookasideListAllocateHits

The number of times an allocation could be satisfied by one of the small nonpaged lookaside lists.

SmallPagedLookasideListAllocateHits

The number of times an allocation could be satisfied by one of the small-paged lookaside lists.

MmSystemCachePage

The number of pages of the system cache in physical memory.

PagedPoolPage

The number of pages of paged pool in physical memory.

SystemDriverPage

The number of pages of pageable device driver code and static data in physical memory.

FastReadNoWait

The number of asynchronous fast read operations.

FastReadWait

The number of synchronous fast read operations.

FastReadResourceMiss

The number of fast read operations not possible because of resource conflicts.

FastReadNotPossible

The number of fast read operations not possible because file system intervention required.

FastMdlReadNoWait

The number of asynchronous fast read operations requesting a Memory Descriptor List (MDL) for the data.

FastMdlReadWait

The number of synchronous fast read operations requesting an MDL for the data.

FastMdlReadResourceMiss

The number of synchronous fast read operations requesting an MDL for the data not possible because of resource conflicts.

FastMdlReadNotPossible

The number of synchronous fast read operations requesting an MDL for the data not possible because file system intervention required.

MapDataNoWait

The number of asynchronous data map operations.

MapDataWait

The number of synchronous data map operations.

MapDataNoWaitMiss

The number of asynchronous data map operations that incurred page faults.

MapDataWaitMiss

The number of synchronous data map operations that incurred page faults.

PinMappedDataCount

The number of requests to pin mapped data.

PinReadNoWait

The number of asynchronous requests to pin mapped data.

PinReadWait

The number of synchronous requests to pin mapped data.

PinReadNoWaitMiss

The number of asynchronous requests to pin mapped data that incurred page faults when pinning the data.

PinReadWaitMiss

The number of synchronous requests to pin mapped data that incurred page faults when pinning the data.

CopyReadNoWait

The number of asynchronous copy read operations.

CopyReadWait

The number of synchronous copy read operations.

CopyReadNoWaitMiss

The number of asynchronous copy read operations that incurred page faults when reading from the cache.

CopyReadWaitMiss

The number of synchronous copy read operations that incurred page faults when reading from the cache.

MdlReadNoWait

The number of synchronous read operations requesting an MDL for the cached data.

MdlReadWait

The number of synchronous read operations requesting an MDL for the cached data.

MdlReadNoWaitMiss

The number of synchronous read operations requesting an MDL for the cached data that incurred page faults.

MdlReadWaitMiss

The number of synchronous read operations requesting an MDL for the cached data that incurred page faults.

ReadAheadIos

The number of read ahead operations performed in anticipation of sequential access.

LazyWriteIos

The number of write operations initiated by the Lazy Writer.

LazyWritePages

The number of pages written by the Lazy Writer.

DataFlushes

The number of cache flushes in response to flush requests.

DataPages

The number of cache pages flushed in response to flush requests.

ContextSwitches

The number of context switches.

FirstLevelTbFills

The number of first level translation buffer fills.

SecondLevelTbFills

The number of second level translation buffer fills.

SystemCalls

The number of system calls executed.

Remarks

Slightly longer descriptions of many of the members of this structure can be found in the Win32 documentation for the NT Performance Counters.

SystemTimeOfDayInformation

```
typedef struct _SYSTEM_TIME_OF_DAY_INFORMATION { // Information Class 3
  LARGE_INTEGER BootTime;
  LARGE_INTEGER CurrentTime;
  LARGE_INTEGER TimeZoneBias;
  ULONG CurrentTimeZoneId;
} SYSTEM_TIME_OF_DAY_INFORMATION, *PSYSTEM_TIME_OF_DAY_INFORMATION;
```

Members

BootTime

The time when the system was booted in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601).

CurrentTime

The current time of day in the standard time format.

TimeZoneBias

The difference, in 100-nanosecond units, between Coordinated Universal Time (UTC) and local time.

CurrentTimeZoneId

A numeric identifier for the current time zone.

Remarks

None.

SystemProcessesAndThreadsInformation

```
typedef struct _SYSTEM_PROCESSES { // Information Class 5
  ULONG NextEntryDelta;
  ULONG ThreadCount;
  ULONG Reserved1[6];
  LARGE_INTEGER CreateTime;
  LARGE_INTEGER UserTime;
  LARGE_INTEGER KernelTime;
  UNICODE_STRING ProcessName;
  KPRIORITY BasePriority;
  ULONG ProcessId;
  ULONG InheritedFromProcessId;
  ULONG HandleCount;
  ULONG Reserved2[2];
  LLLONG PrivatePageCount;
  VM_COUNTERS VmCounters;
  IO_COUNTERS IoCounters; // Windows 2000 only
  SYSTEM_THREADS Threads[1];
} SYSTEM_PROCESSES, *PSYSTEM_PROCESSES;

typedef struct _SYSTEM_THREADS {
  LARGE_INTEGER KernelTime;
  LARGE_INTEGER UserTime;
  LARGE_INTEGER CreateTime;
  ULONG WaitTime;
  PVOID StartAddress;
  CLIENT_ID ClientId;
  KPRIORITY Priority;
  KPRIORITY BasePriority;
  ULONG ContextSwitchCount;
  THREAD_STATE State;
```

```
  KWAIT_REASON WaitReason;
} SYSTEM_THREADS, *PSYSTEM_THREADS;
```

Members

NextEntryDelta

The offset, from the start of this structure, to the next entry. A `NextEntryDelta` of zero indicates that this is the last structure in the returned data.

ThreadCount

The number of threads in the process.

CreateTime

The creation time of the process in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601).

UserTime

The sum of the time spent executing in user mode by the threads of the process, measured in units of 100-nanoseconds.

KernelTime

The sum of the time spent executing in kernel mode by the threads of the process, measured in units of 100-nanoseconds.

ProcessName

The name of the process, normally derived from the name of the executable file used to create the process.

BasePriority

The default base priority for the threads of the process.

ProcessId

The process identifier of the process.

InheritedFromProcessId

The process identifier of the process from which handles and/or address space was inherited.

HandleCount

The number of handles opened by the process.

VmCounters

Statistics on the virtual memory usage of the process. `VM_COUNTERS` is defined thus in ntddk.h:

```
typedef struct _VM_COUNTERS {
  ULONG PeakVirtualSize;
  ULONG VirtualSize;
  ULONG PageFaultCount;
  ULONG PeakWorkingSetSize;
  ULONG WorkingSetSize;
  ULONG QuotaPeakPagedPoolUsage;
  ULONG QuotaPagedPoolUsage;
  ULONG QuotaPeakNonPagedPoolUsage;
  ULONG QuotaNonPagedPoolUsage;
  ULONG PagefileUsage;
  ULONG PeakPagefileUsage;
} VM_COUNTERS, *PVM_COUNTERS;
```

IoCounters

Statistics on the I/O operations of the process. This information is only present in Windows 2000. `IO_COUNTERS` is defined thus:

```
typedef struct _IO_COUNTERS {
  LARGE_INTEGER ReadOperationCount;
  LARGE_INTEGER WriteOperationCount;
  LARGE_INTEGER OtherOperationCount;
  LARGE_INTEGER ReadTransferCount;
  LARGE_INTEGER WriteTransferCount;
  LARGE_INTEGER OtherTransferCount;
} IO_COUNTERS, *PIO_COUNTERS;
```

PrivatePageCount

The current size, in bytes, of the private (non-shared) pages of the process. Normally has the same value as the VMCounters member PagefileUsage.

Threads

An array of `SYSTEM_THREADS` structures describing the threads of the process. The number of elements in the array is available in the `ThreadCount` member.

The members of `SYSTEM_THREADS` follow.

KernelTime

The time spent executing in kernel mode, measured in units of 100-nanoseconds.

UserTime

The time spent executing in user mode, measured in units of 100-nanoseconds.

CreateTime

The creation time of the thread in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601).

WaitTime

The time at which the thread last entered a wait state, measured in clock ticks since system boot.

StartAddress

The start address of the thread.

ClientId

The client identifier of the thread, comprising a process identifier and a thread identifier.

Priority

The priority of the thread.

BasePriority

The base priority of the thread.

ContextSwitchCount

The number of context switches incurred by the thread.

State

The execution state of the thread. Permitted values are drawn from the enumeration `THREAD_STATE`.

```
typedef enum {
  StateInitialized,
  StateReady,
  StateRunning,
  StateStandby,
  StateTerminated,
  StateWait,
  StateTransition,
  StateUnknown
} THREAD_STATE;
```

WaitReason

An indication of the reason for a wait. Some possible values are defined in the enumeration `KWAIT_REASON`, but other values may also be used.

```
typedef enum _KWAIT_REASON {
  Executive,
  FreePage,
  PageIn,
  PoolAllocation,
  DelayExecution,
  Suspended,
  UserRequest,
  WrExecutive,
  WrFreePage,
  WrPageIn,
  WrPoolAllocation,
  WrDelayExecution,
  WrSuspended,
  WrUserRequest,
  WrEventPair,
  WrQueue,
  WrLpcReceive,
  WrLpcReply,
  WrVirtualMemory,
  WrPageOut,
  WrRendezvous,
  Spare2,
  Spare3,
  Spare4,
  Spare5,
  Spare6,
  WrKernel
} KWAIT_REASON;
```

Remarks

The format of the data returned to the SystemInformation buffer is a sequence of `SYSTEM_PROCESSES` structures, chained together via the `NextEntryDelta` member. The `Threads` member of each `SYSTEM_PROCESSES` structure is an array of `ThreadCount SYSTEM_THREADS` structures. The end of the process chain is marked by a `NextEntryDelta` value of zero.

The Process Status API (PSAPI) function `EnumProcesses` uses this information class to obtain a list of the process identifier in the system.

A demonstration of the use of this information class to implement a subset of the Tool Help Library appears in Example 1.1.

The addition of the `IoCounters` member to `SYSTEM_PROCESSES` structure in Windows 2000 has the consequence that Windows NT 4.0 applications that access the `Threads` member fail when run under Windows 2000; for example, the pstat.exe resource kit utility suffers from this problem.

SystemCallCounts

```
typedef struct _SYSTEM_CALLS_INFORMATION { // Information Class 6
  ULONG Size;
  ULONG NumberOfDescriptorTables;
```

```
  ULONG NumberOfRoutinesInTable[1];
  // ULONG CallCounts[];
} SYSTEM_CALLS_INFORMATION, *PSYSTEM_CALLS_INFORMATION;
```

Members

Size

The size in bytes of the returned information.

NumberOfDescriptorTables

The number of system service dispatch descriptor tables for which information is available.

NumberOfRoutinesInTable

An array of the count of routines in each table.

Remarks

Information on the number of calls to each system service is only gathered if the "checked" version of the kernel is used and memory is allocated by the creator of the table to hold the counts.

The counts of calls to each system service follow the array `NumberOfRoutinesInTable`.

SystemConfigurationInformation

```
typedef struct _SYSTEM_CONFIGURATION_INFORMATION { // Information Class 7
  ULONG DiskCount;
  ULONG FloppyCount;
  ULONG CdRomCount;
  ULONG TapeCount;
  ULONG SerialCount;
  ULONG ParallelCount;
} SYSTEM_CONFIGURATION_INFORMATION, *PSYSTEM_CONFIGURATION_INFORMATION;
```

Members

DiskCount

The number of hard disk drives in the system.

FloppyCount

The number of floppy disk drives in the system.

CdRomCount

The number of CD-ROM drives in the system.

TapeCount

The number of tape drives in the system.

SerialCount

The number of serial ports in the system.

ParallelCount

The number of parallel ports in the system.

Remarks

This information is a subset of the information available to device drivers by calling `IoGetConfigurationInformation`.

SystemProcessorTimes

```
typedef struct _SYSTEM_PROCESSOR_TIMES { // Information Class 8
  LARGE_INTEGER IdleTime;
  LARGE_INTEGER KernelTime;
  LARGE_INTEGER UserTime;
  LARGE_INTEGER DpcTime;
  LARGE_INTEGER InterruptTime;
  ULONG InterruptCount;
} SYSTEM_PROCESSOR_TIMES, *PSYSTEM_PROCESSOR_TIMES;
```

Members

IdleTime

The idle time, measured in units of 100-nanoseconds, of the processor.

KernelTime

The time the processor spent executing in kernel mode, measured in units of 100-nanoseconds.

UserTime

The time the processor spent executing in user mode, measured in units of 100-nanoseconds.

DpcTime

The time the processor spent executing deferred procedure calls, measured in units of 100-nanoseconds.

InterruptTime

The time the processor spent executing interrupt routines, measured in units of 100-nanoseconds.

InterruptCount

The number of interrupts serviced by the processor.

Remarks

An array of structures is returned, one per processor.

SystemGlobalFlag

```
typedef struct _SYSTEM_GLOBAL_FLAG { // Information Class 9
  ULONG GlobalFlag;
} SYSTEM_GLOBAL_FLAG, *PSYSTEM_GLOBAL_FLAG;
```

Members

GlobalFlag

A bit array of flags that control various aspects of the behavior of the kernel.

Remarks

This information class can be both queried and set. `SeDebugPrivilege` is required to set the flags. Some flags are used only at boot time and subsequent changes have no effect. Some flags have an effect only when using a "checked" kernel.

The flags recognized by the "gflags" resource kit utility are:

```
FLG_STOP_ON_EXCEPTION          0x00000001
FLG_SHOW_LDR_SNAPS             0x00000002
FLG_DEBUG_INITIAL_COMMAND      0x00000004
FLG_STOP_ON_HUNG_GUI           0x00000008
FLG_HEAP_ENABLE_TAIL_CHECK     0x00000010
FLG_HEAP_ENABLE_FREE_CHECK     0x00000020
FLG_HEAP_VALIDATE_PARAMETERS   0x00000040
FLG_HEAP_VALIDATE_ALL          0x00000080
FLG_POOL_ENABLE_TAIL_CHECK     0x00000100
FLG_POOL_ENABLE_FREE_CHECK     0x00000200
FLG_POOL_ENABLE_TAGGING        0x00000400
FLG_HEAP_ENABLE_TAGGING        0x00000800
FLG_USER_STACK_TRACE_DB        0x00001000
FLG_KERNEL_STACK_TRACE_DB      0x00002000
FLG_MAINTAIN_OBJECT_TYPELIST   0x00004000
FLG_HEAP_ENABLE_TAG_BY_DLL     0x00008000
FLG_IGNORE_DEBUG_PRIV          0x00010000
FLG_ENABLE_CSRDEBUG            0x00020000
FLG_ENABLE_KDEBUG_SYMBOL_LOAD  0x00040000
```

```
    FLG_DISABLE_PAGE_KERNEL_STACKS 0x00080000
    FLG_HEAP_ENABLE_CALL_TRACING   0x00100000
    FLG_HEAP_DISABLE_COALESCING    0x00200000
    FLG_ENABLE_CLOSE_EXCEPTIONS    0x00400000
    FLG_ENABLE_EXCEPTION_LOGGING   0x00800000
    FLG_ENABLE_DBGPRINT_BUFFERING  0x08000000
```

SystemModuleInformation

```
typedef struct _SYSTEM_MODULE_INFORMATION { // Information Class 11
  ULONG Reserved[2];
  PVOID Base;
  ULONG Size;
  ULONG Flags;
  USHORT Index;
  USHORT Unknown;
  USHORT LoadCount;
  USHORT ModuleNameOffset;
  CHAR ImageName[256];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;
```

Members

Base

The base address of the module.

Size

The size of the module.

Flags

A bit array of flags describing the state of the module.

Index

The index of the module in the array of modules.

Unknown

Normally contains zero; interpretation unknown.

LoadCount

The number of references to the module.

ModuleNameOffset

The offset to the final filename component of the image name.

ImageName

The filepath of the module.

Remarks

The data returned to the `SystemInformation` buffer is a `ULONG` count of the number of modules followed immediately by an array of `SYSTEM_MODULE_INFORMATION`.

The system modules are the Portable Executable (PE) format files loaded into the kernel address space (ntoskrnl.exe, hal.dll, device drivers, and so on) and ntdll.dll.

The PSAPI function `EnumDeviceDrivers` uses this information class to obtain a list of the device drivers in the system. It is also used by the PSAPI functions `GetDeviceDriverFileName` and `GetDeviceDriverBaseName`.

The code in Example 1.3 uses this information class.

SystemLockInformation

```
typedef struct _SYSTEM_LOCK_INFORMATION { // Information Class 12
  PVOID Address;
  USHORT Type;
  USHORT Reserved1;
  ULONG ExclusiveOwnerThreadId;
  ULONG ActiveCount;
  ULONG ContentionCount;
  ULONG Reserved2[2];
  ULONG NumberOfSharedWaiters;
  ULONG NumberOfExclusiveWaiters;
} SYSTEM_LOCK_INFORMATION, *PSYSTEM_LOCK_INFORMATION;
```

Members

Address

The address of the `ERESOURCE` structure.

Type

The type of the lock. This is always `RTL_RESOURCE_TYPE (1)`.

ExclusiveOwnerThreadId

The thread identifier of the owner of the resource if the resource is owned exclusively, otherwise zero.

ActiveCount

The number of threads granted access to the resource.

ContentionCount

The number of times a thread had to wait for the resource.

NumberOfSharedWaiters

The number of threads waiting for shared access to the resource.

NumberOfExclusiveWaiters

The number of threads waiting for exclusive access to the resource.

Remarks

The data returned to the `SystemInformation` buffer is a `ULONG` count of the number of locks followed immediately by an array of `SYSTEM_LOCK_INFORMATION`.

The locks reported on by this information class are only available to kernel mode code. The locks support multiple reader single writer functionality and are known as "resources." They are initialized by the routine `ExInitializeResourceLite` and are documented in the DDK.

SystemHandleInformation

```
typedef struct _SYSTEM_HANDLE_INFORMATION { // Information Class 16
  ULONG ProcessId;
  UCHAR ObjectTypeNumber;
  UCHAR Flags; // 0x01 = PROTECT_FROM_CLOSE, 0x02 = INHERIT
  USHORT Handle;
  PVOID Object;
  ACCESS_MASK GrantedAccess;
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;
```

Members

ProcessId

The process identifier of the owner of the handle.

ObjectTypeNumber

A number which identifies the type of object to which the handle refers. The number can be translated to a name by using the information returned by ZwQueryObject.

Flags

A bit array of flags that specify properties of the handle.

Handle

The numeric value of the handle.

Object

The address of the kernel object to which the handle refers.

GrantedAccess

The access to the object granted when the handle was created.

Remarks

The data returned to the `SystemInformation` buffer is a `ULONG` count of the number of handles followed immediately by an array of `SYSTEM_HANDLE_INFORMATION`.

Examples of the use of this information class to implement utilities that list the open handles of processes appear in Example 1.2 and Example 2.1 in Chapter 2, "Objects, Object Directories, and Symbolic Links."

SystemObjectInformation

```
typedef struct _SYSTEM_OBJECT_TYPE_INFORMATION { // Information Class 17
  ULONG NextEntryOffset;
  ULONG ObjectCount;
  ULONG HandleCount;
  ULONG TypeNumber;
  ULONG InvalidAttributes;
  GENERIC_MAPPING GenericMapping;
  ACCESS_MASK ValidAccessMask;
  POOL_TYPE PoolType;
  UCHAR Unknown;
  UNICODE_STRING Name;
} SYSTEM_OBJECT_TYPE_INFORMATION, *PSYSTEM_OBJECT_TYPE_INFORMATION;

typedef struct _SYSTEM_OBJECT_INFORMATION {
  ULONG NextEntryOffset;
  PVOID Object;
  ULONG CreatorProcessId;
  USHORT Unknown;
  USHORT Flags;
  ULONG PointerCount;
  ULONG HandleCount;
  ULONG PagedPoolUsage;
  ULONG NonPagedPoolUsage;
  ULONG ExclusiveProcessId;
  PSECURITY_DESCRIPTOR SecurityDescriptor;
  UNICODE_STRING Name;
} SYSTEM_OBJECT_INFORMATION, *PSYSTEM_OBJECT_INFORMATION;
```

Members

NextEntryOffset

The offset from the start of the `SystemInformation` buffer to the next entry.

ObjectCount

The number of objects of this type in the system.

HandleCount

The number of handles to objects of this type in the system.

TypeNumber

A number that identifies this object type.

InvalidAttributes

A bit mask of the `OBJ_Xxx` attributes that are not valid for objects of this type. The defined attributes are

```
OBJ_INHERIT
OBJ_PERMANENT
OBJ_EXCLUSIVE
OBJ_CASE_INSENSITIVE
OBJ_OPENIF
OBJ_OPENLINK
OBJ_KERNEL_HANDLE     // Windows 2000 only
```

GenericMapping

The mapping of generic access rights to specific access rights for this object type.

ValidAccessMask

The valid specific access rights for this object type.

PoolType

The type of pool from which this object type is allocated (paged or nonpaged).

Unknown

Interpretation unknown.

Name

A name that identifies this object type.

The members of `SYSTEM_OBJECT_INFORMATION` follow.

NextEntryOffset

The offset from the start of the `SystemInformation` buffer to the next entry.

Object

The address of the object.

CreatorProcessId

The process identifier of the creator of the object.

Unknown

Normally contains zero; interpretation unknown.

Flags

A bit array of flags that specify properties of the object. Observed values include:

```
SINGLE_HANDLE_ENTRY     0x40
DEFAULT_SECURITY_QUOTA  0x20
PERMANENT               0x10
EXCLUSIVE               0x08
CREATOR_INFO            0x04
KERNEL_MODE             0x02
```

PointerCount

The number of pointer references to the object.

HandleCount

The number of handle references to the object.

PagedPoolUsage

The amount of paged pool used by the object.

NonPagedPoolUsage

The amount of nonpaged pool used by the object.

ExclusiveProcessId

The process identifier of the owner of the object if it was created for exclusive use (by specifying `OBJ_EXCLUSIVE`).

SecurityDescriptor

The security descriptor for the object.

Name

The name of the object.

Remarks

This information class is only available if `FLG_MAINTAIN_OBJECT_TYPELIST` was set in `NtGlobalFlags` at boot time.

The format of the data returned to the `SystemInformation` buffer is a sequence of `SYSTEM_OBJECT_TYPE_INFORMATION` structures, chained together via the `NextEntryOffset` member. Immediately following the name of the object type is a sequence of `SYSTEM_OBJECT_INFORMATION` structures, which are chained together via the `NextEntryOffset` member. The ends of both the object type chain and the object chain are marked by a `NextEntryOffset` value of zero.

The use of this information class to implement a utility that lists the open handles of processes appears in Example 1.2.

SystemPagefileInformation

```
typedef struct _SYSTEM_PAGEFILE_INFORMATION { // Information Class 18
  ULONG NextEntryOffset;
  ULONG CurrentSize;
  ULONG TotalUsed;
  ULONG PeakUsed;
  UNICODE_STRING FileName;
} SYSTEM_PAGEFILE_INFORMATION, *PSYSTEM_PAGEFILE_INFORMATION;
```

Members

NextEntryOffset

The offset from the start of the `SystemInformation` buffer to the next entry.

CurrentSize

The current size in pages of the page file.

TotalUsed

The number of pages in the page file that are in use.

PeakUsed

The peak number of pages in the page file that have been in use.

FileName

The filepath of the page file.

Remarks

None.

SystemInstructionEmulationCounts

```
typedef struct _SYSTEM_INSTRUCTION_EMULATION_INFORMATION { // Info Class 19
  ULONG SegmentNotPresent;
  ULONG TwoByteOpcode;
  ULONG ESprefix;
  ULONG CSprefix;
  ULONG SSprefix;
  ULONG DSprefix;
  ULONG FSPrefix;
  ULONG GSprefix;
  ULONG OPER32prefix;
  ULONG ADDR32prefix;
  ULONG INSB;
  ULONG INSW;
  ULONG OUTSB;
  ULONG OUTSW;
  ULONG PUSHFD;
  ULONG POPFD;
  ULONG INTnn;
  ULONG INTO;
  ULONG IRETD;
  ULONG INBimm;
  ULONG INWimm;
  ULONG OUTBimm;
  ULONG OUTWimm;
  ULONG INB;
  ULONG INW;
  ULONG OUTB;
  ULONG OUTW;
  ULONG LOCKprefix;
  ULONG REPNEprefix;
  ULONG REPprefix;
  ULONG HLT;
  ULONG CLI;
  ULONG STI;
  ULONG GenericInvalidOpcode;
} SYSTEM_INSTRUCTION_EMULATION_INFORMATION, *PSYSTEM_INSTRUCTION_EMULATION_INFORMAT
```

Remarks

The members of this structure are the number of times that particular instructions had to be emulated for virtual DOS machines. The prefix opcodes do not themselves require emulation, but they may prefix an opcode that does require emulation.

SystemCacheInformation

```
typedef struct _SYSTEM_CACHE_INFORMATION { // Information Class 21
  ULONG SystemCacheWsSize;
  ULONG SystemCacheWsPeakSize;
  ULONG SystemCacheWsFaults;
  ULONG SystemCacheWsMinimum;
  ULONG SystemCacheWsMaximum;
```

```
  ULONG TransitionSharedPages;
  ULONG TransitionSharedPagesPeak;
  ULONG Reserved[2];
} SYSTEM_CACHE_INFORMATION, *PSYSTEM_CACHE_INFORMATION;
```

Members

SystemCacheWsSize

The size in bytes of the system working set.

SystemCacheWsPeakSize

The peak size in bytes of the system working set.

SystemCacheWsFaults

The number of page faults incurred by the system working set.

SystemCacheWsMinimum

The minimum desirable size in pages of the system working set.

SystemCacheWsMaximum

The maximum desirable size in pages of the system working set.

TransitionSharedPages

The sum of the number of pages in the system working set and the number of shared pages on the Standby list. This value is only valid in Windows 2000.

TransitionSharedPagesPeak

The peak of the sum of the number of pages in the system working set and the number of shared pages on the Standby list. This value is only valid in Windows 2000.

Remarks

This information class can be both queried and set. When setting, only the `SystemCacheWsMinimum` and `SystemCacheWsMaximum` values are used.

SystemPoolTagInformation

```
typedef struct _SYSTEM_POOL_TAG_INFORMATION { // Information Class 22
  CHAR Tag[4];
  ULONG PagedPoolAllocs;
  ULONG PagedPoolFrees;
  ULONG PagedPoolUsage;
  ULONG NonPagedPoolAllocs;
```

```
  ULONG NonPagedPoolFrees;
  ULONG NonPagedPoolUsage;
} SYSTEM_POOL_TAG_INFORMATION, *PSYSTEM_POOL_TAG_INFORMATION;
```

Members

Tag

The four character tag string identifying the contents of the pool allocation.

PagedPoolAllocs

The number of times a block was allocated from paged pool with this tag.

PagedPoolFrees

The number of times a block was deallocated to paged pool with this tag.

PagedPoolUsage

The number of bytes of paged pool used by blocks with this tag.

NonPagedPoolAllocs

The number of times a block was allocated from nonpaged pool with this tag.

NonPagedPoolFrees

The number of times a block was deallocated to nonpaged pool with this tag.

NonPagedPoolUsage

The number of bytes of nonpaged pool used by blocks with this tag.

Remarks

This information class is only available if `FLG_POOL_ENABLE_TAGGING` was set in `NtGlobalFlags` at boot time.

The data returned to the SystemInformation buffer is a `ULONG` count of the number of tags followed immediately by an array of `SYSTEM_POOL_TAG_INFORMATION`.

The data returned by this information class is displayed by the "poolmon" utility.

SystemProcessorStatistics

```
typedef struct _SYSTEM_PROCESSOR_STATISTICS { // Information Class 23
  ULONG ContextSwitches;
  ULONG DpcCount;
```

```
  ULONG DpcRequestRate;
  ULONG TimeIncrement;
  ULONG DpcBypassCount;
  ULONG ApcBypassCount;
} SYSTEM_PROCESSOR_STATISTICS, *PSYSTEM_PROCESSOR_STATISTICS;
```

Members

ContextSwitches

The number of context switches performed by the processor.

DpcCount

The number of deferred procedure calls (DPC) that have been added to the processor's DPC queue.

DpcRequestRate

The number of DPCs that have been added to the processor's DPC queue since the last clock tick.

TimeIncrement

The number of 100-nanosecond units between ticks of the system clock.

DpcBypassCount

The number of DPC interrupts that have been avoided.

ApcBypassCount

The number of kernel APC interrupts that have been avoided.

Remarks

An array of structures is returned, one per processor.

The `ReturnLength` information is not set correctly (always contains zero).

SystemDpcInformation

```
typedef struct _SYSTEM_DPC_INFORMATION { // Information Class 24
  ULONG Reserved;
  ULONG MaximumDpcQueueDepth;
  ULONG MinimumDpcRate;
  ULONG AdjustDpcThreshold;
  ULONG IdealDpcRate;
} SYSTEM_DPC_INFORMATION, *PSYSTEM_DPC_INFORMATION;
```

Members

MaximumDpcQueueDepth

The maximum depth that the DPC queue should attain. If this depth is exceeded and no DPCs are active, a DPC interrupt is requested.

MinimumDpcRate

The minimum rate at which DPCs should be requested. If the current request rate is lower and no DPCs are active, a DPC interrupt is requested.

AdjustDpcThreshold

A parameter that affects the interval between retuning of the DPC parameters.

IdealDpcRate

The ideal rate at which DPCs should be requested. If the current rate is higher, measures are taken to tune the DPC parameters (for example, by adjusting the maximum DPC queue depth).

Remarks

This information class can be both queried and set. `SeLoadDriverPrivilege` is required to set the values.

These parameters only affect `MediumInportance` and `HighImportance` DPCs.

The `ReturnLength` information is not set correctly (always contains zero).

SystemLoadImage

```
typedef struct _SYSTEM_LOAD_IMAGE { // Information Class 26
  UNICODE_STRING ModuleName;
  PVOID ModuleBase;
  PVOID Unknown;
  PVOID EntryPoint;
  PVOID ExportDirectory;
} SYSTEM_LOAD_IMAGE, *PSYSTEM_LOAD_IMAGE;
```

Members

ModuleName

The full path in the native NT format of the module to load. Required on input.

ModuleBase

The base address of the module. Valid on output.

ModuleSection

Pointer to a data structure describing the loaded module. Valid on output.

EntryPoint

The address of the entry point of the module. Valid on output.

ExportDirectory

The address of the export directory of the module. Valid on output.

Remarks

This information class can only be set. Rather than setting any information (in a narrow sense of "setting"), it performs the operation of loading a module into the kernel address space and returns information on the loaded module.

After loading the module, `MmPageEntireDriver` (documented in the DDK) is called to make the entire module pageable. The module entry point is not called.

This information class is valid only when **ZwSetSystemInformation** is invoked from kernel mode.

SystemUnloadImage

```
typedef struct _SYSTEM_UNLOAD_IMAGE { // Information Class 27
  PVOID ModuleBase;
} SYSTEM_UNLOAD_IMAGE, *PSYSTEM_UNLOAD_IMAGE;
```

Members

ModuleSection

Pointer to the data structure describing the loaded module.

Remarks

This information class can only be set. Rather than setting any information (in a narrow sense of "setting"), it performs the operation of unloading a module from the kernel address space.

Even if the module is a device driver, the `DriverUnload` routine is not called.

This information class is only valid when **ZwSetSystemInformation** is invoked from kernel mode.

SystemTimeAdjustment

```
typedef struct _SYSTEM_QUERY_TIME_ADJUSTMENT { // Information Class 28
```

```
  ULONG TimeAdjustment;
  ULONG MaximumIncrement;
  BOOLEAN TimeSynchronization;
} SYSTEM_QUERY_TIME_ADJUSTMENT, *PSYSTEM_QUERY_TIME_ADJUSTMENT;

typedef struct _SYSTEM_SET_TIME_ADJUSTMENT { // Information Class 28
  ULONG TimeAdjustment;
  BOOLEAN TimeSynchronization;
} SYSTEM_SET_TIME_ADJUSTMENT, *PSYSTEM_SET_TIME_ADJUSTMENT;
```

Members

TimeAdjustment

The number of 100-nanosecond units added to the time-of-day clock at each clock tick if time adjustment is enabled.

MaximumIncrement

The maximum number of 100-nanosecond units between clock ticks. Also the number of 100-nanosecond units per clock tick for kernel intervals measured in clock ticks.

TimeSynchronization

A boolean specifying that time adjustment is enabled when true.

Remarks

This information class can be both queried and set. `SeSystemtimePrivilege` is required to set the values. The structures for querying and setting values are different.

The `ReturnLength` information is not set correctly (always contains zero).

SystemCrashDumpInformation

```
typedef struct _SYSTEM_CRASH_DUMP_INFORMATION { // Information Class 32
  HANDLE CrashDumpSectionHandle;
  HANDLE Unknown; // Windows 2000 only
} SYSTEM_CRASH_DUMP_INFORMATION, *PSYSTEM_CRASH_DUMP_INFORMATION;
```

Members

CrashDumpSectionHandle

A handle to the crash dump section.

ModuleSection

A handle to an unknown object. This information is only present in Windows 2000.

Remarks

If a crash dump section exists, a new handle to the section is created for the current process and returned in `CrashDumpSectionHandle`; otherwise, `CrashDumpSectionHandle` contains zero.

In Windows 2000, `SeCreatePagefilePrivilege` is required to query the values.

SystemExceptionInformation

```
typedef struct _SYSTEM_EXCEPTION_INFORMATION { // Information Class 33
  ULONG AlignmentFixupCount;
  ULONG ExceptionDispatchCount;
  ULONG FloatingEmulationCount;
  ULONG Reserved;
} SYSTEM_EXCEPTION_INFORMATION, *PSYSTEM_EXCEPTION_INFORMATION;
```

Members

AlignmentFixupCount

The numbers of times data alignment had to be fixed up since the system booted.

ExceptionDispatchCount

The number of exceptions dispatched since the system booted.

FloatingEmulationCount

The number of times floating point instructions had to be emulated since the system booted.

Remarks

None.

SystemCrashDumpStateInformation

```
typedef struct _SYSTEM_CRASH_DUMP_STATE_INFORMATION { // Information Class 34
  ULONG CrashDumpSectionExists;
  ULONG Unknown; // Windows 2000 only
} SYSTEM_CRASH_DUMP_STATE_INFORMATION, *PSYSTEM_CRASH_DUMP_STATE_INFORMATION;
```

Members

CrashDumpSectionExists

A boolean indicating whether a crash dump section exists.

ModuleSection

Interpretation unknown. This information is only present in Windows 2000.

Remarks

In Windows 2000, this information class can also be set if `SeCreatePagefilePrivilege` is enabled.

SystemKernelDebuggerInformation

```
typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION { // Information Class 35
  BOOLEAN DebuggerEnabled;
  BOOLEAN DebuggerNotPresent;
} SYSTEM_KERNEL_DEBUGGER_INFORMATION, *PSYSTEM_KERNEL_DEBUGGER_INFORMATION;
```

Members

DebuggerEnabled

A boolean indicating whether kernel debugging has been enabled or not.

DebuggerNotPresent

A boolean indicating whether contact with a remote debugger has been established or not.

Remarks

None.

SystemContextSwitchInformation

```
typedef struct _SYSTEM_CONTEXT_SWITCH_INFORMATION { // Information Class 36
  ULONG ContextSwitches;
  ULONG ContextSwitchCounters[11];
} SYSTEM_CONTEXT_SWITCH_INFORMATION, *PSYSTEM_CONTEXT_SWITCH_INFORMATION;
```

Members

ContextSwitches

The number of context switches.

ContextSwitchCounters

Normally contains zeroes; interpretation unknown.

Remarks

The resource kit utility "`kernprof`" claims to display the context switch counters (if the "-x" option is specified), but it only expects nine `ContextSwitchCounters` rather than eleven. It displays the information thus:

```
    Context Switch Information
      Find any processor         0
      Find last processor        0
      Idle any processor         0
      Idle current processor     0
      Idle last processor        0
      Preempt any processor      0
      Preempt current processor  0
      Preempt last processor     0
      Switch to idle             0
```

SystemRegistryQuotaInformation

```
typedef struct _SYSTEM_REGISTRY_QUOTA_INFORMATION { // Information Class 37
  ULONG RegistryQuota;
  ULONG RegistryQuotaInUse;
  ULONG PagedPoolSize;
} SYSTEM_REGISTRY_QUOTA_INFORMATION, *PSYSTEM_REGISTRY_QUOTA_INFORMATION;
```

Members

RegistryQuota

The number of bytes of paged pool that the registry may use.

RegistryQuotaInUse

The number of bytes of paged pool that the registry is using.

PagedPoolSize

The size in bytes of the paged pool.

Remarks

This information class can be both queried and set. `SeIncreaseQuotaPrivilege` is required to set the values. When setting, only the `RegistryQuota` value is used.

SystemLoadAndCallImage

```
typedef struct _SYSTEM_LOAD_AND_CALL_IMAGE { // Information Class 38
  UNICODE_STRING ModuleName;
} SYSTEM_LOAD_AND_CALL_IMAGE, *PSYSTEM_LOAD_AND_CALL_IMAGE;
```

Members

ModuleName

The full path in the native NT format of the module to load.

Remarks

This information class can only be set. Rather than setting any information (in a narrow sense of "setting"), it performs the operation of loading a module into the kernel address space and calling its entry point.

The entry point routine is expected to be a `__stdcall` routine taking two parameters (consistent with the `DriverEntry` routine of device drivers); the call arguments are two zeroes.

If the entry point routine returns a failure code, the module is unloaded.

Unlike ZwLoadDriver, which loads the module in the context of the system process, ZwSetSystemInformation loads the module and invokes the entry point in the context of the current process.

SystemPrioritySeparation

```
typedef struct _SYSTEM_PRIORITY_SEPARATION { // Information Class 39
  ULONG PrioritySeparation;
} SYSTEM_PRIORITY_SEPARATION, *PSYSTEM_PRIORITY_SEPARATION;
```

Members

PrioritySeparation

> A value that affects the scheduling quantum period of the foreground application. In Windows NT 4.0, `PrioritySeparation` takes a value between zero and two (the higher the value, the longer the quantum period). In Windows 2000, the low order six bits of `PrioritySeparation` are used to configure the scheduling quantum.

Remarks

None.

SystemTimeZoneInformation

```
typedef struct _SYSTEM_TIME_ZONE_INFORMATION { // Information Class 44
  LONG Bias;
  WCHAR StandardName[32];
  SYSTEMTIME StandardDate;
  LONG StandardBias;
  WCHAR DaylightName[32];
  SYSTEMTIME DaylightDate;
  LONG DaylightBias;
} SYSTEM_TIME_ZONE_INFORMATION, *PSYSTEM_TIME_ZONE_INFORMATION;
```

Members

Bias

The difference, in minutes, between Coordinated Universal Time (UTC) and local time.

StandardName

The name of the timezone when daylight saving time is not in effect.

StandardDate

A `SYSTEMTIME` structure specifying when daylight saving time ends.

StandardBias

The difference, in minutes, between UTC and local time when daylight saving time is not in effect.

DaylightName

The name of the timezone when daylight saving time is in effect.

DaylightDate

A `SYSTEMTIME` structure specifying when daylight saving time starts.

DaylightBias

The difference, in minutes, between UTC and local time when daylight saving time is in effect.

Remarks

This structure is identical to the `TIME_ZONE_INFORMATION` structure returned by the Win32 function `GetTimeZoneInformation`.

SystemLookasideInformation

```
typedef struct _SYSTEM_LOOKASIDE_INFORMATION { // Information Class 45
  USHORT Depth;
  USHORT MaximumDepth;
  ULONG TotalAllocates;
  ULONG AllocateMisses;
  ULONG TotalFrees;
  ULONG FreeMisses;
  POOL_TYPE Type;
  ULONG Tag;
  ULONG Size;
} SYSTEM_LOOKASIDE_INFORMATION, *PSYSTEM_LOOKASIDE_INFORMATION;
```

Members

Depth

The current depth of the lookaside list.

MaximumDepth

The maximum depth of the lookaside list.

TotalAllocates

The total number of allocations made from the list.

AllocateMisses

The number of times the lookaside list was empty and a normal allocation was needed.

TotalFrees

The total number of allocations made from the list.

FreeMisses

The number of times the lookaside list was full and a normal deallocation was needed.

Type

The type of pool from which the memory for the lookaside list is allocated. Possible values are drawn from the enumeration `POOL_TYPE`:

```
typedef enum _POOL_TYPE {
  NonPagedPool,
  PagedPool,
  NonPagedPoolMustSucceed,
  DontUseThisType,
  NonPagedPoolCacheAligned,
  PagedPoolCacheAligned,
  NonPagedPoolCacheAlignedMustS,
  MaxPoolType
  NonPagedPoolSession = 32,
  PagedPoolSession,
  NonPagedPoolMustSucceedSession,
  DontUseThisTypeSession,
  NonPagedPoolCacheAlignedSession,
  PagedPoolCacheAlignedSession,
  NonPagedPoolCacheAlignedMustSSession
} POOL_TYPE;
```

Tag

The tag identifying allocations from the lookaside list

Size

The size of the blocks on the lookaside list.

Remarks

An array of structures are returned, one per lookaside list. The number of structures can be obtained by dividing the `ReturnLength` by the size of the structure.

The lookaside lists reported on by this information class are only available to kernel mode code. Their purpose is to speed the allocation and deallocation of blocks of memory from paged and nonpaged pool. A nonpaged lookaside list is initialized by the routine `ExInitializeNPagedLookasideList`.

Lookaside lists are documented in the DDK.

SystemSetTimeSlipEvent

```
typedef struct _SYSTEM_SET_TIME_SLIP_EVENT { // Information Class 46
  HANDLE TimeSlipEvent;
} SYSTEM_SET_TIME_SLIP_EVENT, *PSYSTEM_SET_TIME_SLIP_EVENT;
```

Members

TimeSlipEvent

A handle to an event object. The handle must grant `EVENT_MODIFY_STATE` access.

Remarks

This information class can only be set. `SeSystemtimePrivilege` is required to set the value. The `TimeSlipEvent` will be signaled when the kernel debugger has caused time to slip by blocking the system clock interrupt.

SystemCreateSession

```
typedef struct _SYSTEM_CREATE_SESSION { // Information Class 47
  ULONG SessionId;
} SYSTEM_CREATE_SESSION, *PSYSTEM_CREATE_SESSION;
```

Members

SessionId

An identifier for the session. Valid on output.

Remarks

This information class can only be set. It creates a Windows Terminal Server session and assigns the session an identifier. This information class is valid only when Windows Terminal Server is running.

In all other cases, the return status is `STATUS_INVALID_SYSTEM_SERVICE`.

SystemDeleteSession

```
typedef struct _SYSTEM_DELETE_SESSION { // Information Class 48
  ULONG SessionId;
} SYSTEM_DELETE_SESSION, *PSYSTEM_DELETE_SESSION;
```

< h3>Members

SessionId

An identifier for the session

Remarks

This information class can only be set. This information class is valid only when Windows Terminal Server is running. In all other cases the return status is `STATUS_INVALID_SYSTEM_SERVICE`.

SystemRangeStartInformation

```
typedef struct _SYSTEM_RANGE_START_INFORMATION { // Information Class 50
  PVOID SystemRangeStart;
} SYSTEM_RANGE_START_INFORMATION, *PSYSTEM_RANGE_START_INFORMATION;
```

Members

SystemRangeStart

The base address of the system (kernel) portion of the virtual address space.

Remarks

None.

SystemVerifierInformation

Format unknown.

Remarks

This information class can be both queried and set. `SeDebugPrivilege` is required to set the values.

This information class queries and sets information maintained by the device driver verifier. The "Driver Verifier" is described in the DDK documentation.

SystemAddVerifier

Format unknown.

Remarks

This information class is only valid when **ZwSetSystemInformation** is invoked from kernel mode.

This information class configures the device driver verifier. The "Driver Verifier" is described in the DDK documentation.

SystemSessionProcessesInformation

```
typedef struct _SYSTEM_SESSION_PROCESSES_INFORMATION { // Information Class 53
  ULONG SessionId;
  ULONG BufferSize;
  PVOID Buffer;
} SYSTEM_SESSION_PROCESSES_INFORMATION, *PSYSTEM_SESSION_PROCESSES_INFORMATION;
```

Members

SessionId

The SessionId for which to retrieve a list of processes and threads.

BufferSize

The size in bytes of the buffer in which to return the list of processes and threads.

Buffer

Points to a caller-allocated buffer or variable that receives the list of processes and threads.

Remarks

Unlike other information classes, this information class uses the `SystemInformation` argument of **ZwQuerySystemInformation** as an input buffer.

The information returned is in the same format as that returned by `SystemProcessesAndThreadsInformation`, but contains information only on the processes in the specified session.

The following information classes are only available in "checked" versions of the kernel.

SystemPoolBlocksInformation

```
typedef struct _SYSTEM_POOL_BLOCKS_INFORMATION { // Info Classes 14 and 15
  ULONG PoolSize;
  PVOID PoolBase;
```

```
  USHORT Unknown;
  ULONG NumberOfBlocks;
  SYSTEM_POOL_BLOCK PoolBlocks[1];
} SYSTEM_POOL_BLOCKS_INFORMATION, *PSYSTEM_POOL_BLOCKS_INFORMATION;

typedef struct _SYSTEM_POOL_BLOCK {
  BOOLEAN Allocated;
  USHORT Unknown;
  ULONG Size;
  CHAR Tag[4];
} SYSTEM_POOL_BLOCK, *PSYSTEM_POOL_BLOCK;
```

Members

PoolSize

The size in bytes of the pool.

PoolBase

The base address of the pool.

ModuleSection

The alignment of the pool; interpretation uncertain.

NumberOfBlocks

The number of blocks in the pool.

PoolBlocks

An array of `SYSTEM_POOL_BLOCK` structures describing the blocks in the pool. The number of elements in the array is available in the `NumberOfBlocks` member.

The members of `SYSTEM_POOL_BLOCK` follow.

Allocated

A boolean indicating whether this is an allocated or free block.

ModuleSection

Interpretation unknown.

Size

The size in bytes of the block.

Tag

The four character tag string identifying the contents of the pool allocation.

Remarks

Information class 14 returns data on the paged pool and information class 15 returns data on the nonpaged pool.

The paged and nonpaged pools reported on by these information classes are only available to kernel mode code. Blocks are allocated from paged and nonpaged pool by the routines `ExAllocatePoolXxx`. The use of pool memory is documented in the DDK.

SystemMemoryUsageInformation

```
typedef struct _SYSTEM_MEMORY_USAGE_INFORMATION { // Info Classes 25 and 29
  ULONG Reserved;
  PVOID EndOfData;
  SYSTEM_MEMORY_USAGE MemoryUsage[1];
} SYSTEM_MEMORY_USAGE_INFORMATION, *PSYSTEM_MEMORY_USAGE_INFORMATION;

typedef struct _SYSTEM_MEMORY_USAGE {
  PVOID Name;
  USHORT Valid;
  USHORT Standby;
  USHORT Modified;
  USHORT PageTables;
} SYSTEM_MEMORY_USAGE, *PSYSTEM_MEMORY_USAGE;
```

Members

EndOfData

A pointer to the end of the valid data in the `SystemInformation` buffer.

MemoryUsage

An array of `SYSTEM_MEMORY_USAGE` structures describing the usage of physical memory. The number of elements in the array is deducible from the `EndOfData` member.

The members of `SYSTEM_MEMORY_USAGE` follow.

Name

The name of the object using the memory. This can be either a Unicode or ANSI string.

Valid

The number of valid pages used by the object. If the object is a process, this is the number of valid private pages.

Standby

The number of pages recently used by the object that are now on the Standby list.

Modified

The number of pages recently used by the object, which are now on the Modified list.

PageTables

The number of pagetable pages used by the object. The only objects that use pagetables are processes. On an Intel platform using large (4-MByte) pages, the pagetables are charged against nonpaged pool rather than processes.

Remarks

Information class 29 does not provide the information on the pages in the Standby and Modified lists.

There is no indication of whether the name is a Unicode or ANSI string other than the string data itself (for example, if every second byte is zero, the string must be Unicode).

Information class 25 is able to account for the use of almost all the physical memory in the system. The difference between sum of the `Valid`, `Standby` and `Modified` pages and the `NumberOfPhysicalPages` (returned by the `SystemBasicInformation` class) is normally close to the number of pages on the Free and Zeroed memory lists.

Example 1.1: A Partial ToolHelp Library Implementation

```
#include "ntdll.h"
#include <tlhelp32.h>
#include <stdio.h>

struct ENTRIES {
  ULONG Offset;
  ULONG Count;
  ULONG Index;
  ENTRIES() : Offset(0), Count(0), Index(0) {}
  ENTRIES(ULONG m, ULONG n) : Offset(m), Count(n), Index(0) {}
};

enum EntryType {
  ProcessType,
  ThreadType,
  MaxType
};


NT::PSYSTEM_PROCESSES GetProcessesAndThreads()
{
  ULONG n = 0x100;
  NT::PSYSTEM_PROCESSES sp = new NT::SYSTEM_PROCESSES[n];

  while (NT::ZwQuerySystemInformation(
                  NT::SystemProcessesAndThreadsInformation,
                  sp, n * sizeof *sp, 0)
      == STATUS_INFO_LENGTH_MISMATCH)
```

```
    delete [ ] sp, sp = new NT::SYSTEM_PROCESSES[n = n * 2];

  return sp;
}

ULONG ProcessCount(NT::PSYSTEM_PROCESSES sp)
{
  ULONG n = 0;

  bool done = false;

  for (NT::PSYSTEM_PROCESSES p = sp; !done;
     p = NT::PSYSTEM_PROCESSES(PCHAR(p) + p->NextEntryDelta))
     n++, done = p->NextEntryDelta == 0;

  return n;
}

ULONG ThreadCount(NT::PSYSTEM_PROCESSES sp)
{
  ULONG n = 0;

  bool done = false;

  for (NT::PSYSTEM_PROCESSES p = sp; !done;
     p = NT::PSYSTEM_PROCESSES(PCHAR(p) + p->NextEntryDelta))
     n += p->ThreadCount, done = p->NextEntryDelta == 0;

  return n;
}

VOID AddProcesses(PPROCESSENTRY32 pe, NT::PSYSTEM_PROCESSES sp)
{
  bool done = false;

  for (NT::PSYSTEM_PROCESSES p = sp; !done;
     p = NT::PSYSTEM_PROCESSES(PCHAR(p) + p->NextEntryDelta)) {

    pe->dwSize = sizeof *pe;
    pe->cntUsage = 0;
    pe->th32ProcessID = p->ProcessId;
    pe->th32DefaultHeapID = 0;
    pe->th32ModuleID = 0;
    pe->cntThreads = p->ThreadCount;
    pe->th32ParentProcessID = p->InheritedFromProcessId;
    pe->pcPriClassBase = p->BasePriority;
    pe->dwFlags = 0;
    sprintf(pe->szExeFile, "%.*ls",
        p->ProcessName.Length / 2, p->ProcessName.Buffer);

    pe++;

    done = p->NextEntryDelta == 0;
  }
}

VOID AddThreads(PTHREADENTRY32 te, NT::PSYSTEM_PROCESSES sp)
{
  bool done = false;

  for (NT::PSYSTEM_PROCESSES p = sp; !done;
     p = NT::PSYSTEM_PROCESSES(PCHAR(p) + p->NextEntryDelta)) {

    for (ULONG i = 0; i < p->ThreadCount; i++) {
```

```
      te->dwSize = sizeof *te;
      te->cntUsage = 0;
      te->th32ThreadID = DWORD(p->Threads[i].ClientId.UniqueThread);
      te->th32OwnerProcessID = p->ProcessId;
      te->tpBasePri = p->Threads[i].BasePriority;
      te->tpDeltaPri = p->Threads[i].Priority
              - p->Threads[i].BasePriority;
      te->dwFlags = 0;

      te++;
    }

    done = p->NextEntryDelta == 0;
  }
}

template<class T>
BOOL GetEntry(HANDLE hSnapshot, T entry, bool first, EntryType type)
{
  ENTRIES *entries = (ENTRIES*)MapViewOfFile(hSnapshot, FILE_MAP_WRITE,
                        0, 0, 0);
  if (entries == 0) return FALSE;

  BOOL rv = TRUE;

  entries[type].Index = first ? 0 : entries[type].Index + 1;

  if (entries[type].Index >= entries[type].Count)
    SetLastError(ERROR_NO_MORE_FILES), rv = FALSE;

  if (entry->dwSize < sizeof *entry)
    SetLastError(ERROR_INSUFFICIENT_BUFFER), rv = FALSE;

  if (rv)
    *entry = T(PCHAR(entries)+entries[type].Offset)[entries[type].Index];

  UnmapViewOfFile(entries);

  return rv;
}


HANDLE
WINAPI
CreateToolhelp32Snapshot(DWORD flags, DWORD)
{
  NT::PSYSTEM_PROCESSES sp =
    (flags & (TH32CS_SNAPPROCESS | TH32CS_SNAPTHREAD))
      ? GetProcessesAndThreads() : 0;

  ENTRIES entries[MaxType];
  ULONG n = sizeof entries;

  if (flags & TH32CS_SNAPPROCESS) {
    entries[ProcessType] = ENTRIES(n, ProcessCount(sp));
    n += entries[ProcessType].Count * sizeof (PROCESSENTRY32);
  }
  if (flags & TH32CS_SNAPTHREAD) {
    entries[ThreadType] = ENTRIES(n, ThreadCount(sp));
    n += entries[ThreadType].Count * sizeof (THREADENTRY32);
  }

  SECURITY_ATTRIBUTES sa = {sizeof sa, 0, (flags & TH32CS_INHERIT) != 0};
```

```
  HANDLE hMap = CreateFileMapping(HANDLE(0xFFFFFFFF), &sa,
                 PAGE_READWRITE | SEC_COMMIT, 0, n, 0);

  ENTRIES *p = (ENTRIES*)MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, 0);

  for (int i = 0; i < MaxType; i++) p[i] = entries[i];

  if (flags & TH32CS_SNAPPROCESS)
    AddProcesses(PPROCESSENTRY32(PCHAR(p) + entries[ProcessType].Offset),
          sp);
  if (flags & TH32CS_SNAPTHREAD)
    AddThreads(PTHREADENTRY32(PCHAR(p) + entries[ThreadType].Offset),
          sp);

  UnmapViewOfFile(p);

  if (sp) delete [ ] sp;

  return hMap;
}

BOOL
WINAPI
Thread32First(HANDLE hSnapshot, PTHREADENTRY32 te)
{
  return GetEntry(hSnapshot, te, true, ThreadType);
}

BOOL
WINAPI
Thread32Next(HANDLE hSnapshot, PTHREADENTRY32 te)
{
  return GetEntry(hSnapshot, te, false, ThreadType);
}

BOOL
WINAPI
Process32First(HANDLE hSnapshot, PPROCESSENTRY32 pe)
{
  return GetEntry(hSnapshot, pe, true, ProcessType);
}

BOOL
WINAPI
Process32Next(HANDLE hSnapshot, PPROCESSENTRY32 pe)
{
  return GetEntry(hSnapshot, pe, false, ProcessType);
}
```

ZwQuerySystemInformation with an information class of `SystemProcessesAndThreadsInformation` returns a superset of the information concerning processes and threads that is available via the ToolHelp library (if it were implemented in Windows NT 4.0). Example 1.1 uses this information class to implement a subset of the ToolHelp library; the remaining functions of the ToolHelp library are addressed in later chapters.

The Win32 function `CreateToolhelp32Snapshot` returns a handle to a snapshot of the processes and threads (and modules and heaps) in the system. The Win32 documentation states that this handle (and the snapshot itself) is freed by calling `CloseHandle`. **ZwQuerySystemInformation** also returns a "snapshot," but this snapshot is just data in a caller-supplied buffer. To implement the documented behavior of `CreateToolhelp32Snapshot`, it is necessary to encapsulate the information returned by

ZwQuerySystemInformation in a kernel object so that `CloseHandle` can free it.

The only suitable kernel object is a section object (known as a file mapping object by Win32). The idea is to create a paging-file backed section object and then map a view of this section into the address space so that the information returned from **ZwQuerySystemInformation** can be copied to it. The view is then unmapped so that closing the section handle will free the snapshot (mapped views prevent the section object from being deleted).

The routines that return information from the snapshot must then just map the section, copy the relevant data to the caller-supplied buffer, and unmap the section.

Example 1.2: Listing Open Handles of a Process

```
#include "ntdll.h"
#include <stdlib.h>
#include <stdio.h>
#include <vector>
#include <map>

#pragma warning(disable:4786) // identifier was truncated in the debug info

struct OBJECTS_AND_TYPES {
  std::map<ULONG, NT::PSYSTEM_OBJECT_TYPE_INFORMATION, std::less<ULONG> >
    types;
  std::map<PVOID, NT::PSYSTEM_OBJECT_INFORMATION, std::less<PVOID> >
    objects;
};

std::vector<NT::SYSTEM_HANDLE_INFORMATION> GetHandles()
{
  ULONG n;
  PULONG p = new ULONG[n = 0x100];

  while (NT::ZwQuerySystemInformation(NT::SystemHandleInformation,
                    p, n * sizeof *p, 0)
      == STATUS_INFO_LENGTH_MISMATCH)

    delete [ ] p, p = new ULONG[n *= 2];

  NT::PSYSTEM_HANDLE_INFORMATION h = NT::PSYSTEM_HANDLE_INFORMATION(p + 1);

  return std::vector<NT::SYSTEM_HANDLE_INFORMATION>(h, h + *p);
}

OBJECTS_AND_TYPES GetObjectsAndTypes()
{
  ULONG n;
  PCHAR p = new CHAR[n = 0x1000];

  while (NT::ZwQuerySystemInformation(NT::SystemObjectInformation,
                    p, n * sizeof *p, 0)
      == STATUS_INFO_LENGTH_MISMATCH)

    delete [ ] p, p = new CHAR[n *= 2];

  OBJECTS_AND_TYPES oats;

  for (NT::PSYSTEM_OBJECT_TYPE_INFORMATION
     t = NT::PSYSTEM_OBJECT_TYPE_INFORMATION(p); ;
     t = NT::PSYSTEM_OBJECT_TYPE_INFORMATION(p + t->NextEntryOffset)) {
```

```
    oats.types[t->TypeNumber] = t;

    for (NT::PSYSTEM_OBJECT_INFORMATION
       o = NT::PSYSTEM_OBJECT_INFORMATION(PCHAR(t->Name.Buffer)
                     + t->Name.MaximumLength); ;
       o = NT::PSYSTEM_OBJECT_INFORMATION(p + o->NextEntryOffset)) {

      oats.objects[o->Object] = o;

      if (o->NextEntryOffset == 0) break;
    }
    if (t->NextEntryOffset == 0) break;
  }

  return oats;
}

int main(int argc, char *argv[ ])
{
  if (argc == 1) return 0;

  ULONG pid = strtoul(argv[1], 0, 0);

  OBJECTS_AND_TYPES oats = GetObjectsAndTypes();

  std::vector<NT::SYSTEM_HANDLE_INFORMATION> handles = GetHandles();

  NT::SYSTEM_OBJECT_INFORMATION defobj = {0};

  printf("Object  Hnd Access Fl Atr #H  #P Type      Name\n");

  for (std::vector<NT::SYSTEM_HANDLE_INFORMATION>::iterator
     h = handles.begin(); h != handles.end(); h++) {

    if (h->ProcessId == pid) {

      NT::PSYSTEM_OBJECT_TYPE_INFORMATION
        t = oats.types[h->ObjectTypeNumber];
      NT::PSYSTEM_OBJECT_INFORMATION
        o = oats.objects[h->Object];

      if (o == 0) o = &defobj;

      printf("%p %04hx %6lx %2x %3hx %3ld %4ld %-14.*S %.*S\n",
          h->Object, h->Handle, h->GrantedAccess, int(h->Flags),
          o->Flags, o->HandleCount, o->PointerCount,
          t->Name.Length, t->Name.Buffer,
          o->Name.Length, o->Name.Buffer);
    }
  }
  return 0;
}
```

Example 1.2 assumes that the `NtGlobalFlag FLG_MAINTAIN_OBJECT_TYPELIST` was set at boot time. An alternative method of obtaining a list of open handles using a combination of **ZwQuerySystemInformation** and **ZwQueryObject** appears in Chapter 2, "Objects, Object Directories, and Symbolic Links," in Example 2.1.

The program uses the address of the kernel object to which a handle refers to correlate the information returned by the information classes `SystemHandleInformation` and

`SystemObjectInformation`; a Standard Template Library (STL) map is used for this purpose.

The list of handles in the system is scanned for handles owned by a particular process identifier, and then information about the handle and the object to which it refers is displayed.

ZwQuerySystemEnvironmentValue

ZwQuerySystemEnvironmentValue queries the value of a system environment variable stored in the non-volatile (CMOS) memory of the system.

```
NTSYSAPI
NTSTATUS
NTAPI
ZwQuerySystemEnvironmentValue(
  IN PUNICODE_STRING Name,
  OUT PVOID Value,
  IN ULONG ValueLength,
  OUT PULONG ReturnLength OPTIONAL
  );
```

Parameters

Name

The name of system environment value to be queried.

Value

Points to a caller-allocated buffer or variable that receives the requested system environment value.

ValueLength

The size in bytes of `Value`.

ReturnLength

Optionally points to a variable that receives the number of bytes actually returned to `Value`. If `ValueLength` is too small to contain the available data, the variable is set to the number of bytes required for the available data. If this information is not needed by the caller, `ReturnLength` may be specified as a null pointer.

Return Value

Returns `STATUS_SUCCESS` or an error status, such as `STATUS_PRIVILEGE_NOT_HELD`, `STATUS_BUFFER_OVERFLOW`, or `STATUS_UNSUCCESSFUL`.

Related Win32 Functions

None.

Remarks

`SeSystemEnvironmentPrivilege` is required to query system environment values.

The information returned in `Buffer` is an array of `WCHAR`. The `ReturnLength` value contains the length of the string in bytes.

ZwQuerySystemEnvironmentValue queries environment values stored in CMOS. The standard Hardware Abstraction Layer (HAL) for the Intel platform only supports one environment value, "`LastKnownGood`," which takes the values "`TRUE`" and "`FALSE`." It is queried by writing 0xb to port 0x70 and reading from port 0x71. A value of zero is interpreted as "`FALSE`," other values as "`TRUE`."

ZwSetSystemEnvironmentValue

ZwSetSystemEnvironmentValue sets the value of a system environment variable stored in the non-volatile (CMOS) memory of the system.

```
NTSYSAPI
NTSTATUS
NTAPI
ZwSetSystemEnvironmentValue(
  IN PUNICODE_STRING Name,
  IN PUNICODE_STRING Value
  );
```

Parameters

Name

The name of system environment value to be set.

Value

The value to be set.

Return Value

Returns `STATUS_SUCCESS` or an error status, such as `STATUS_PRIVILEGE_NOT_HELD` or `STATUS_UNSUCCESSFUL`.

Related Win32 Functions

None.

Remarks

`SeSystemEnvironmentPrivilege` is required to set system environment values.

ZwSetSystemEnvironmentValue sets environment values stored in CMOS. The standard HAL for the Intel platform only supports one environment value, "LastKnownGood," which takes the values "TRUE" and "FALSE." It is set by writing 0xb to port 0x70 and writing 0 (for "FALSE") or 1 (for "TRUE") to port 0x71.

ZwShutdownSystem

ZwShutdownSystem shuts down the system.

```
NTSYSAPI
NTSTATUS
NTAPI
ZwShutdownSystem(
  IN SHUTDOWN_ACTION Action
  );
```

Parameters

Action

The action to be performed after shutdown. Permitted values are drawn from the enumeration SHUTDOWN_ACTION.

```
typedef enum _SHUTDOWN_ACTION {
  ShutdownNoReboot,
  ShutdownReboot,
  ShutdownPowerOff
} SHUTDOWN_ACTION;
```

Return Value

Returns STATUS_SUCCESS or an error status, such as STATUS_PRIVILEGE_NOT_HELD.

Related Win32 Functions

ExitWindows(Ex), InitiateSystemShutdown.

Remarks

SeShutdownPrivilege is required to shut down the system.

User-mode applications and services are not informed of the shutdown (drivers of devices that have registered for shutdown notification by calling IoRegisterShutdownNotification are informed).

The system must have hardware support for power-off if the power-off action is to be used successfully.

ZwSystemDebugControl

ZwSystemDebugControl performs a subset of the operations available to a kernel mode debugger.

```
NTSYSAPI
NTSTATUS
NTAPI
ZwSystemDebugControl(
  IN DEBUG_CONTROL_CODE ControlCode,
  IN PVOID InputBuffer OPTIONAL,
  IN ULONG InputBufferLength,
  OUT PVOID OutputBuffer OPTIONAL,
  IN ULONG OutputBufferLength,
  OUT PULONG ReturnLength OPTIONAL
  );
```

Parameters

ControlCode

The control code for operation to be performed. Permitted values are drawn from the enumeration `DEBUG_CONTROL_CODE`.

```
typedef enum _DEBUG_CONTROL_CODE {
  DebugGetTraceInformation = 1,
  DebugSetInternalBreakpoint,
  DebugSetSpecialCall,
  DebugClearSpecialCalls,
  DebugQuerySpecialCalls,
  DebugDbgBreakPoint
} DEBUG_CONTROL_CODE;
```

InputBuffer

Points to a caller-allocated buffer or variable that contains the data required to perform the operation. This parameter can be null if the `ControlCode` parameter specifies an operation that does not require input data.

InputBufferLength

The size in bytes of `InputBuffer`.

OutputBuffer

Points to a caller-allocated buffer or variable that receives the operation's output data. This parameter can be null if the `ControlCode` parameter specifies an operation that does not produce output data.

OutputBufferLength

The size in bytes of `OutputBuffer`.

ReturnLength

Optionally points to a variable that receives the number of bytes actually returned to `OutputBuffer`. If this information is not needed, `ReturnLength` may be a null pointer.

Return Value

Returns `STATUS_SUCCESS` or an error status, such as `STATUS_PRIVILEGE_NOT_HELD`, `STATUS_INVALID_INFO_CLASS` or `STATUS_INFO_LENGTH_MISMATCH`.

Related Win32 Functions

None.

Remarks

`SeDebugPrivilege` is required to use **ZwSystemDebugControl** in Windows 2000.

ZwSystemDebugControl allows a process to perform a subset of the functions available to a kernel mode debugger.

The system should be booted from a configuration that has the boot.ini "`/DEBUG`" (or equivalent) option enabled; otherwise a kernel debugger variable needed for the correct operation of internal breakpoints is not initialized.

The data structures used by **ZwSystemDebugControl** are defined in windbgkd.h (included with the Platform SDK). An up-to-date copy of this file is needed to compile the code in Examples 1.3 and 1.4. One of the structures used by **ZwSystemDebugControl** includes a union that has grown over time, and **ZwSystemDebugControl** checks that the input/output buffers are large enough to hold the largest member of the union.

DebugGetTraceInformation

```
typedef struct _DBGKD_GET_INTERNAL_BREAKPOINT { // DebugGetTraceInformation
  DWORD_PTR BreakpointAddress;
  DWORD Flags;
  DWORD Calls;
  DWORD MaxCallsPerPeriod;
  DWORD MinInstructions;
  DWORD MaxInstructions;
  DWORD TotalInstructions;
} DBGKD_GET_INTERNAL_BREAKPOINT, *PDBGKD_GET_INTERNAL_BREAKPOINT;

#define DBGKD_INTERNAL_BP_FLAG_COUNTONLY 0x01 // don't count instructions
#define DBGKD_INTERNAL_BP_FLAG_INVALID  0x02 // disabled BP
#define DBGKD_INTERNAL_BP_FLAG_SUSPENDED 0x04 // temporarily suspended
#define DBGKD_INTERNAL_BP_FLAG_DYING   0x08 // kill on exit
```

`DebugGetTraceInformation` does not require an InputBuffer and returns an array of `DBGKD_GET_INTERNAL_BREAKPOINT` structures in the output buffer, one for each of the internal breakpoints set.

Instruction counting counts the instructions from the breakpoint until the return from the routine

containing the breakpoint. Ideally, the breakpoint should be placed at the beginning of a routine. The user mode debugger (windbg, cdb, ntsd) command "wt" performs user mode instruction counting.

If instruction counting is enabled, `MinInstructions` contains the minimum number of instructions encountered when executing the routine, `MaxInstructions` contains the maximum, and `TotalInstructions` contains the total number of instructions executed by all invocations of the routine (since the breakpoint was inserted).

`Calls` is the number of times the breakpoint has been encountered.

`Flags` indicates whether instruction counting is enabled and whether the breakpoint has been suspended.

DebugSetInternalBreakpoint

```
typedef struct _DBGKD_MANIPULATE_STATE {
  DWORD ApiNumber;
  WORD ProcessorLevel;
  WORD Processor;
  DWORD ReturnStatus;
  union {
    DBGKD_READ_MEMORY ReadMemory;
    DBGKD_WRITE_MEMORY WriteMemory;
    DBGKD_READ_MEMORY64 ReadMemory64;
    DBGKD_WRITE_MEMORY64 WriteMemory64;
    DBGKD_GET_CONTEXT GetContext;
    DBGKD_SET_CONTEXT SetContext;
    DBGKD_WRITE_BREAKPOINT WriteBreakPoint;
    DBGKD_RESTORE_BREAKPOINT RestoreBreakPoint;
    DBGKD_CONTINUE Continue;
    DBGKD_CONTINUE2 Continue2;
    DBGKD_READ_WRITE_IO ReadWriteIo;
    DBGKD_READ_WRITE_IO_EXTENDED ReadWriteIoExtended;
    DBGKD_QUERY_SPECIAL_CALLS QuerySpecialCalls;
    DBGKD_SET_SPECIAL_CALL SetSpecialCall;
    DBGKD_SET_INTERNAL_BREAKPOINT SetInternalBreakpoint;
    DBGKD_GET_INTERNAL_BREAKPOINT GetInternalBreakpoint;
    DBGKD_GET_VERSION GetVersion;
    DBGKD_BREAKPOINTEX BreakPointEx;
    DBGKD_PAGEIN PageIn;
    DBGKD_READ_WRITE_MSR ReadWriteMsr;
  } u;
} DBGKD_MANIPULATE_STATE, *PDBGKD_MANIPULATE_STATE;

typedef struct _DBGKD_SET_INTERNAL_BREAKPOINT { // DebugSetInternalBreakpoint
  DWORD_PTR BreakpointAddress;
  DWORD Flags;
} DBGKD_SET_INTERNAL_BREAKPOINT, *PDBGKD_SET_INTERNAL_BREAKPOINT;
```

`DebugSetInternalBreakpoint` does not require an `OutputBuffer` and expects the `InputBuffer` to point to a `DBGKD_MANIPULATE_STATE` structure. The only values in this structure that are required are the two values in the `DBGKD_SET_INTERNAL_BREAKPOINT` structure. `InputBufferLength` is the size of the `DBGKD_MANIPULATE_STATE` structure.

`BreakpointAddress` is the address of the breakpoint. If a breakpoint already exists at this address, the `Flags` are used to manipulate the breakpoint, otherwise a new breakpoint is established. Breakpoints are deleted by setting the `DBGKD_INTERNAL_BP_FLAG_INVALID` flag and are temporarily

suspended by setting the DBGKD_INTERNAL_BP_FLAG_SUSPENDED flag. The counting or non-counting nature of the breakpoint can be controlled by setting or clearing the DBGKD_INTERNAL_BP_FLAG_COUNTONLY flag.

Breakpoints can be set at any address, but if the address is not at the start of an instruction then an STATUS_ILLEGAL_INSTRUCTION exception may be raised resulting in a system crash. The intention is that breakpoints should be set at the start of routines but, particularly if instruction counting is disabled, this is not essential.

DebugSetSpecialCall

```
typedef struct _DBGKD_SET_SPECIAL_CALL { // DebugSetSpecialCall
  DWORD SpecialCall;
} DBGKD_SET_SPECIAL_CALL, *PDBGKD_SET_SPECIAL_CALL;
```

DebugSetSpecialCall does not require an OutputBuffer and expects the InputBuffer to point to a DBGKD_MANIPULATE_STATE structure. The only value in this structure that is required is the value in the DBGKD_SET_SPECIAL_CALL structure. InputBufferLength must be four rather than the size of the DBGKD_MANIPULATE_STATE structure—this is a bug.

"Special Calls" are routines that should be treated specially when counting the instructions executed by some routine. The special calls set by the kernel debugger are:

```
HAL!@KfLowerIrql@4
HAL!@KfReleaseSpinLock@8
HAL!@HalRequestSoftwareInterrupt@4
NTOSKRNL!SwapContext
NTOSKRNL!@KiUnlockDispatcherDatabase@4
```

Whether the members of this list are necessary or sufficient to ensure correct operation of the instruction counting feature is difficult to say.

DebugClearSpecialCalls

DebugClearSpecialCalls requires neither an InputBuffer nor an OutputBuffer. It clears the list of special calls.

DebugQuerySpecialCalls

```
typedef struct _DBGKD_QUERY_SPECIAL_CALLS { // DebugQuerySpecialCalls
  DWORD NumberOfSpecialCalls;
  // DWORD SpecialCalls[ ];
} DBGKD_QUERY_SPECIAL_CALLS, *PDBGKD_QUERY_SPECIAL_CALLS;
```

DebugQuerySpecialCalls does not require an InputBuffer and expects the OutputBuffer to point to a buffer large enough to hold a DBGKD_MANIPULATE_STATE structure and an array of DWORDs, one per special call. It returns a list of the special calls.

DebugDbgBreakPoint

DebugDbgBreakPoint requires neither an InputBuffer nor an OutputBuffer. If the kernel debugger

is enabled it causes a kernel mode debug break point to be executed. This debug control code is only valid in Windows 2000.

The code in Examples 1.3 and 1.4 demonstrates how to set internal breakpoints and get trace information.

Example 1.3: Setting an Internal Breakpoint

```
#include "ntdll.h"
#include "windbgkd.h"
#include <imagehlp.h>
#include <stdlib.h>

void LoadModules()
{
  ULONG n;
  NT::ZwQuerySystemInformation(NT::SystemModuleInformation,
                &n, 0, &n);
  PULONG p = new ULONG[n];
  NT::ZwQuerySystemInformation(NT::SystemModuleInformation,
                p, n * sizeof *p, 0);

  NT::PSYSTEM_MODULE_INFORMATION module
    = NT::PSYSTEM_MODULE_INFORMATION(p + 1);

  for (ULONG i = 0; i < *p; i++)
    SymLoadModule(0, 0, module[i].ImageName,
           module[i].ImageName + module[i].ModuleNameOffset,
           ULONG(module[i].Base), module[i].Size);

  delete [ ] p;
}

DWORD GetAddress(PSTR expr)
{
  PCHAR s;
  ULONG n = strtoul(expr, &s, 16);

  if (*s == 0) return n;

  IMAGEHLP_SYMBOL symbol;

  symbol.SizeOfStruct = sizeof symbol;
  symbol.MaxNameLength = sizeof symbol.Name;

  return SymGetSymFromName(0, expr, &symbol) == TRUE ? symbol.Address : 0;
}

void SetSpecialCall(DWORD addr)
{
  DBGKD_MANIPULATE_STATE op = {0};
  op.u.SetSpecialCall.SpecialCall = addr;

  NT::ZwSystemDebugControl(NT::DebugSetSpecialCall, &op, 4, 0, 0, 0);
}

void SetSpecialCalls()
{
  DBGKD_MANIPULATE_STATE op[4];

  NT::ZwSystemDebugControl(NT::DebugQuerySpecialCalls,
```

```
                0, 0, op, sizeof op, 0);

  if (op[0].u.QuerySpecialCalls.NumberOfSpecialCalls == 0) {
    SetSpecialCall(GetAddress("HAL!KfLowerIrql"));
    SetSpecialCall(GetAddress("HAL!KfReleaseSpinLock"));
    SetSpecialCall(GetAddress("HAL!HalRequestSoftwareInterrupt"));
    SetSpecialCall(GetAddress("NTOSKRNL!SwapContext"));
    SetSpecialCall(GetAddress("NTOSKRNL!KiUnlockDispatcherDatabase"));
  }
}

int main(int argc, char *argv[ ])
{
  if (argc < 2) return 0;

  NT:: SYSTEM_KERNEL_DEBUGGER_INFORMATION kd;

  NT::ZwQuerySystemInformation(NT::SystemKernelDebuggerInformation,
                &kd, sizeof kd, 0);
  if (kd.DebuggerEnabled == FALSE) return 0;

  EnablePrivilege(SE_DEBUG_NAME);

  SymInitialize(0, 0, FALSE);
  SymSetOptions(SymGetOptions() | SYMOPT_DEFERRED_LOADS);

  LoadModules();

  SetSpecialCalls();

  DBGKD_MANIPULATE_STATE op = {0};
  op.u.SetInternalBreakpoint.BreakpointAddress = GetAddress(argv[1]);
  op.u.SetInternalBreakpoint.Flags = argc < 3 ? 0 : strtoul(argv[2], 0, 16);

  NT::ZwSystemDebugControl(NT::DebugSetInternalBreakpoint,
              &op, sizeof op, 0, 0, 0);

  return 0;
}
```

If the kernel debugger is not enabled, an important debugger variable is not initialized. Therefore, Example 1.3 first uses **ZwQuerySystemInformation** to check the debugger status and if it is enabled, the program then sets the special calls and creates or updates a breakpoint.

The program also demonstrates how to obtain a list of the kernel modules and their base addresses. This information is needed by the Imagehlp API routines, which are used to translate symbolic names into addresses.

The program assumes that `SymLoadModule` will find the correct symbol files; if this routine finds the wrong symbol files (for example, symbols for a checked rather than free build), a system crash is almost guaranteed.

Example 1.4: Getting Trace Information

```
#include "ntdll.h"
#include "windbgkd.h"
#include <stdio.h>

int main()
```

```
{
  DBGKD_GET_INTERNAL_BREAKPOINT bp[20];
  ULONG n;

  EnablePrivilege(SE_DEBUG_NAME);

  NT::ZwSystemDebugControl(NT::DebugGetTraceInformation,
              0, 0, bp, sizeof bp, &n);

  for (int i = 0; i * sizeof (DBGKD_GET_INTERNAL_BREAKPOINT) < n; i++)

    printf("%lx %lx %ld %ld %ld %ld %ld\n",
        bp[i].BreakpointAddress, bp[i].Flags,
        bp[i].Calls, bp[i].MaxCallsPerPeriod,
        bp[i].MinInstructions, bp[i].MaxInstructions,
        bp[i].TotalInstructions);

  return 0;
}
```

The output produced by Example 1.4 after an internal breakpoint had been set at `NTOSKRNL!NtCreateProcess` was:

```
80193206 0 6 0 19700 21010 121149
```

Therefore, the minimum number of instructions executed by `NtCreateProcess` was 19,700, the maximum number was 21,010, and the average number was about 20,191.