RTX 5.0

Reference Guide

VenturCom, Inc.
Five Cambridge Center
Cambridge, MA 02142

Tel: 617-661-1230
Fax: 617-577-1607
info@vci.com
http://www.vci.com

RTX 5.0 Reference Guide

1-016-10

Table of Contents

Welcome to RTX 5.0

Document ID: 1-016-10

About RTX

VenturCom's Real-time Extension (RTX) adds real-time capabilities to Windows NT and Windows 2000 that are unparalleled in the industry. It offers developers a rich and powerful real-time feature set — all in a familiar Win32-compatible interface. It also provides tools and utilities for building and executing real-time programs, along with tools for measuring and fine tuning the performance of both hardware and software.

In addition to using the real-time interfaces and tools provided by RTX, developers can continue to take advantage of the abundance of products available for Windows NT and Windows 2000. And, with the RTX inter-process communication features, the Win32 run-time environment works seamlessly with the RTX real-time subsystem — enabling the integration of Win32 and real-time functionality.

Experienced real-time developers will value the power of the RTX interface; we suggest that you refer to the topics in the *RTX Overview* in the *RTX User's Guide* for a more detailed description of RTX and a discussion of important design decisions that need to be made in order to fully take advantage of RTX features.

Getting Support

VenturCom offers a number of support options for RTX users, including technical support and the VenturCom Web site.

Note: If you are a customer who purchased direct support, you would have received a Support ID# in the letter that comes with the software. Please have this number available for reference when contacting VenturCom. Users who purchased their product through third parties should contact those parties directly with their questions.

Technical Support

For technical support related to installing and using RTX, VenturCom offers several channels of communication. You can:

- Call technical support at 800-334-8649 between 9:00 AM and 6:00 PM (Eastern Time)
- Email your questions to support@vci.com
- Fax your questions to 617-577-1607

VenturCom Web Site

The VenturCom Customer Support Web page is located at:

http://www.vci.com/tech_support/support_description.html

If you are a customer with a current support contract or a member of the Real-time and Embedded Partner Program, then you should bookmark to the Web page in the Technical Support Area located at:

http://www.vci.com/tech_support/support_login.html

These pages provide electronic access to the latest product releases, documentation, and release notes. With a valid Support ID#, you can access the online problem report database to submit new issues, or to obtain the status of previously reported issues.

Documentation Updates

VenturCom is committed to providing you with the information you need to use our products. From time to time, we may provide documentation updates for our products. Check our CENTer page for updates. While visiting CENTer, check out the various white pages and presentations. CENTer also provides access to several newsgroups. You'll also find free utilities and extensions that you can download.

CHAPTER 1

Introduction to RTX Programming Interfaces

About the RTX Programming Interfaces

RTX provides an essential set of real-time programming interfaces in the Win32 environment. The RTX interfaces are compatible with the Win32 programming interfaces. In addition, RTX provides extensions to Win32 in order to provide a complete set of real-time functions to the application programmer.

RTX application programs can use the real-time extensions in both the Win32 and RTSS environments and programs can use the Win32-supported API in the real-time susbsystem (RTSS) environment. This provides maximum flexibility to the developer. In the Win32 environment, programs can be developed and tested with the real-time extensions using the vast number of Win32 development tools. The same program can be re-linked as an RTSS program and run deterministically.

The application programming interface for RTX is composed of three sets of interfaces:

- Real-Time API (RTAPI)
- Win32-Supported API
- C Run-Time Library-Supported API

See Also

About the Windows NT and Windows 2000 Driver IPC API (RTKAPI)

Real-Time API (RTAPI) Overview

The RTAPI (real-time application programming interface) set is composed of unique interfaces and Win32-based interfaces. All RTAPI interfaces are identified by the real-time prefix "Rt."

Unique real-time interfaces-These functions are new, Win32-modeled extensions that provide essential programming capabilities required for real-time applications. There is no equivalent Win32 function for these RTAPI functions. The function begins with the "Rt" prefix and the interface semantics are modeled on the Win32 programming interface semantics.

An example is **RtAttachInterruptVector**. This function is required for real-time programming, there is no Win32 functional equivalent, and the "Rt" prefix signifies that the function is an RTAPI function.

Win32-based real-time interfaces-These functions are also extensions to the Win32 functions and provide additional real-time programming capability. Unlike the unique functions above, there are similar functions in the Win32 environment; however, they behave differently. The differences are required by the real-time semantics. The function name is prefixed with "Rt" and the interface semantics are compatible with Win32 programming interface semantics.

An example is **RtCreateMutex**. This function is required for real-time programming. There is a Win32 functional equivalent, but it does not behave exactly like the Win32 **CreateMutex** function. The "Rt" prefix signifies that the function is an RTAPI function.

See Chapter 2, *Real-Time API* for detailed descriptions of each function.

Win32-Supported API Overview

The Win32-Supported functions behave identically to the Win32 functions and are supported in the RTSS environment. The function name is not prefixed with "Rt" since the behavior and the calling interface are identical in both environments.

An example is ResumeThread. This function is supported in the RTSS environment; has identical Win32 equivalent; and the function interface is identical to the Win32 function interface.

See Chapter 3, *Win32-Supported API* for detailed descriptions of each function.

C Run-Time Library-Supported API Overview

An extensive set of Microsoft C Run-time library calls is supported in the RTSS environment.

See the Alphabetical List of C Run-Time APIs in Chapter 4, *C Run-Time*.

About the Windows NT and Windows 2000 Driver IPC API (RTKAPI)

The Windows NT and Windows 2000 Driver Inter-Process Communication API (RTKAPI) functions are used to access RTX IPC mechanisms from Windows NT and Windows 2000 kernel device drivers. These calls are analogous to their RTAPI counterparts (e.g., **RtkOpenSemaphore** is analogous to **RtOpenSemaphore**).

You use the RTKAPI functions the same way as the RTAPI functions, but from the Windows NT and Windows 2000 kernel environment. All RTKAPI interface names are prefixed with "Rtk."

The RTKAPI also consists of an include file (RtkApi.h) and a link library (rtx_rtk.lib).

See Chapter 4, *Windows NT and Windows 2000 Driver IPC API (RTKAPI)* for detailed descriptions of each function.

Matrixes of RTX and RTK Functions

The matrixes provide technical information about the RTX and RTK APIs. Matrixes are provided for:

- Real-Time Functions
- Win32-Supported Functions
- C Library-Supported Functions
- Matrix of RTK Functions

Key

The following key explains the "Notes" column in the tables.

Code for Notes	Meaning
1	The priority spectrum of Rt is 0 to 127, whereas the Win32 range is {-15, -2, -1, 0, 1, 2, 15} on Windows NT and Windows 2000.
2	The RTX IPC namespace is separate from the Win32 namespace.
3	The call is supported as both an Rt call and as a Win32 call (e.g., **RtAtoi** and **Atoi**).
4	Not for use in RTDLL.
5	C Run-Time calls not supported in shared RTDLL.
Determ -inistic	"Yes" means the elapsed time for the call is less than 5 microseconds. Deterministic functions in RTSS work at Windows NT blue screens and Windows 2000 stop screens.
*	Deterministic for small input sizes.

Matrix of Real-Time Functions

RTAPI Function Name	Notes	Deterministic?
RtAllocateContiguousMemory		
RtAllocateLockedMemory		
RtAtoi	3	Yes
RtAttachInterruptVector		
RtAttachInterruptVectorEx		
RtAttachShutdownHandler		
RtCancelTimer		Yes
RtCloseHandle		
RtCommitLockHeap		
RtCommitLockProcessHeap		
RtCommitLockStack		
RtCreateEvent	2	
RtCreateMutex	2	
RtCreateProcess		
RtCreateSemaphore	2	
RtCreateSharedMemory		
RtCreateTimer		
RtDeleteTimer		

RTAPI Function Name	Notes	Deterministic?
RtDisableInterrupts		Yes
RtDisablePortIo		
RtEnableInterrupts		Yes
RtEnablePortIo		
RtFreeContiguousMemory		
RtFreeLockedMemory		
RtGetBusDataByOffset		
RtGetClockResolution		Yes
RtGetClockTime		Yes
RtGetClockTimerPeriod		Yes
GetExitCodeProcess		Yes
RtGetPhysicalAddress		
RtGetThreadPriority	1,3	Yes
RtGetTimer		Yes
RtIsInRtss		Yes
RtLockKernel		
RtLockProcess		
RtMapMemory		
RtOpenEvent	2	
RtOpenMutex	2	
RtOpenProcess		Yes
RtOpenSemaphore	2	
RtOpenSharedMemory		
RtPrintf	3	
RtPulseEvent	2	Yes
RtReadPortBufferUchar		Yes
RtReadPortBufferUlong		Yes
RtReadPortBufferUshort		Yes
RtReadPortUchar		Yes
RtReadPortUlong		Yes
RtReadPortUshort		Yes
RtReleaseInterruptVector		
RtReleaseMutex	2	Yes

RTAPI Function Name	Notes	Deterministic?
RtReleaseSemaphore	2	Yes
RtReleaseShutdownHandler		
RtResetEvent	2	Yes
RtSetBusDataByOffset		
RtSetClockTime		Yes
RtSetEvent	2	Yes
RtSetThreadPriority	1,3	Yes
RtSetThreadTimeQuantum		Yes
RtSetTimer		Yes
RtSetTimerRelative		Yes
RtSleepFt		Yes
RtTerminateProcess		
RtTranslateBusAddress		
RtUnlockKernel		
RtLockProcess		
RtUnmapMemory		
RtWaitForMultipleObjects		Yes
RtWaitForSingleObject		Yes
RtWPrintf	3	
RtWritePortBufferUchar		Yes
RtWritePortBufferUlong		Yes
RtWritePortBufferUshort		Yes
RtWritePortUchar		Yes
RtWritePortUlong		Yes
RtWritePortUshort		Yes
RtWtoi	3	Yes

Matrix of Win32-Supported Functions

Win32 Function Name	Notes	Deterministic?
AbnormalTermination		
CloseHandle		
CreateDirectory		
CreateFile		
CreateThread		

Win32 Function Name	Notes	Deterministic?
DeleteCriticalSection		
DeleteFile		
DeviceIoControl		
DllMain		
EnterCriticalSection		Yes
ExitProcess		
ExitThread		
FreeLibrary		
GetCurrentProcessId		Yes
GetCurrentThread		Yes
GetCurrentThreadId		Yes
GetExceptionCode		Yes
GetExceptionInformation		Yes
GetExitCodeThread		Yes
GetLastError		Yes
GetProcAddress		
GetProcessHeap		
GetThreadPriority	1, 3	Yes
HeapAlloc		
HeapCreate		
HeapDestroy		
HeapFree		
HeapReAlloc		
HeapSize		Yes
InitializeCriticalSection		
LeaveCriticalSection		Yes
LoadLibrary		
RaiseException		Yes
ReadFile		
RemoveDirectory		
ResumeThread		Yes
SetFilePointer		
SetLastError		Yes

Win32 Function Name	Notes	Deterministic?
SetThreadPriority	1, 3	Yes
SetUnhandledExceptionFilter		Yes
Sleep		Yes
SuspendThread		Yes
TerminateThread		
TlsAlloc		Yes
TlsFree		Yes
TlsGetValue		Yes
TlsSetValue		Yes
UnhandledExceptionFilter		Yes
WriteFile		

Matrix of C Library-Supported Functions

C Library Function Name	Notes	Deterministic?
abs	5	Yes
acos	5	Yes
asin	5	Yes
atan	5	Yes
atan2	5	Yes
atof	5	Yes
atoi	3 , 5	Yes
atol	5	Yes
bsearch	5	Yes*
calloc	5	
ceil	5	Yes
cos	5	Yes
cosh	5	Yes
difftime	5	Yes
div	5	Yes
exit	5	
exp	5	Yes
fabs	5	Yes
fclose	5	
fflush	5	
fgets	5	
floor	5	Yes

C Library Function Name	Notes	Deterministic?
fmod	5	Yes
fopen	5	
fprintf(stderr)	4, 5	
fputc	5	
fputs	5	
fread	5	
free	5	
frexp	5	Yes
fseek	5	
ftell	5	
fwrite	5	
getc	5	
isalnum	5	Yes
isalpha	5	Yes
iscntrl	5	Yes
isdigit	5	Yes
isgraph	5	Yes
islower	5	Yes
isprint	5	Yes
ispunct	5	Yes
isspace	5	Yes
isupper	5	Yes
iswalnum	5	Yes
iswalpha	5	Yes
iswascii	5	Yes
iswcntrl	5	Yes
iswctype	5	Yes
iswdigit	5	Yes
iswgraph	5	Yes
iswlower	5	Yes
iswprint	5	Yes
iswpunct	5	Yes
iswspace	5	Yes
iswupper	5	Yes
iswxdigit	5	Yes
isxdigit	5	Yes
labs	5	Yes
ldexp	5	Yes

C Library Function Name	Notes	Deterministic?
ldiv	5	Yes
log	5	Yes
log10	5	Yes
longjmp	5	Yes
main	5	
malloc	5	
memchr	5	Yes
memcmp	5	Yes*
memcpy	5	Yes*
memmove	5	Yes*
memset	5	Yes
modf	5	Yes
perror	4, 5	
pow	5	Yes
printf	3,4,5	
putc	5	
putchar	5	
qsort	5	Yes*
rand	5	Yes
realloc	5	
rewind	5	
setjmp	5	Yes
signal	5	
sin	5	Yes
sinh	5	Yes
sprint	5	Yes
sqrt	5	Yes
srand	5	Yes
sscanf	5	
strcat	5	Yes
strchr	5	Yes
strcmp	5	Yes
strcpy	5	Yes*
strcspn	5	Yes
strerror	5	Yes
strlen	5	Yes
strncat	5	Yes
strncmp	5	Yes*

C Library Function Name	Notes	Deterministic?
strncpy	5	Yes*
strpbrk	5	Yes
strrchr	5	Yes
strspn	5	Yes
strstr	5	Yes
strtod	5	Yes
strtok	5	Yes
strtol	5	Yes
strtoul	5	Yes
tan	5	Yes
tanh	5	Yes
tolower	5	Yes
toupper	5	Yes
towlower	5	Yes
towupper	5	Yes
ungetc	5	Yes
va_start	5	Yes
vsprintf	4,5	
wcscat	5	Yes
wcschr	5	Yes
wcscmp	5	Yes
wcscpy	5	Yes
wcscspn	5	Yes
wcsftime	5	Yes
wcslen	5	Yes
wcsncat	5	Yes
wcsncmp	5	Yes*
wcsncpy	5	Yes*
wcspbrk	5	Yes
wcsrchr	5	Yes
wcsspn	5	Yes
wcsstr	5	Yes
wcstod	5	Yes
wcstok	5	Yes
wcstol	5	Yes
wcstoul	5	Yes
wmain	5	
wprintf	3,4,5	
wtof	5	Yes

C Library Function Name	Notes	Deterministic?
wtoi	3,5	Yes
wtol	5	Yes
_controlfp	5	
_fpreset	5	

Matrix of RTK API Functions

The table that follows lists the RTK API functions.

Note: All RTK functions are available only to Windows NT device drivers; they are not available to RTX applications.

RTK API Function Name	Notes
RtkCloseHandle	
RtkCreateEvent	2
RtkCreateMutex	2
RtkCreateSemaphore	2
RtkCreateSharedMemory	
RtkOpenEvent	2
RtkOpenMutex	2
RtkOpenSemaphore	2
RtkOpenSharedMemory	
RtkPulseEvent	2
RtkReleaseMutex	2
RtkReleaseSemaphore	2
RtkResetEvent	2
RtkRtssAttach	
RtkRtssDetach	
RtkSetEvent	2

Functional Groupings of Real-Time and Win32 APIs

Exception Management APIs

Exception Management Real-Time APIs	Exception Management Win32-Supported APIs
RtAttachShutdownHandler RtReleaseShutdownHandler	AbnormalTermination GetExceptionCode GetExceptionInformation RaiseException SetUnhandledExceptionFilter UnhandledExceptionFilter

Clocks and Timers APIs

Clocks and Timers Real-Time APIs	Clocks and Timers Win32-Supported APIs
RtCancelTimer	Sleep
RtCreateTimer	
RtDeleteTimer	
RtGetClockResolution	
RtGetClockTime	
RtGetClockTimerPeriod	
RtGetTimer	
RtSetClockTime	
RtSetTimer	
RtSetTimerRelative	
RtSleepFt	

General Use APIs

General Use Real-Time APIs	General Use Win32-Supported APIs
RtAtoi	CloseHandle
RtCloseHandle	CreateDirectory
RtIsInRtss	CreateFile
RtPrintf	DeleteCriticalSection
RtWPrintf	DeleteFile
RtWtoi	DeviceIoControl
	DllMain
	EnterCriticalSection
	FreeLibrary
	GetLastError
	GetProcAddress
	InitializeCriticalSection
	LeaveCriticalSection
	LoadLibrary
	ReadFile
	RemoveDirectory
	SetFilePointer
	SetLastError
	WriteFile

Interrupt Services APIs

Interrupt Services Real-Time APIs
RtAttachInterruptVector
RtAttachInterruptVectorEx
RtDisableInterrupts
RtEnableInterrupts
RtReleaseInterruptVector

Inter-Process Communication (IPC) APIs

IPC Real-Time APIs
RtCreateEvent
RtCreateMutex
RtCreateSemaphore
RtCreateSharedMemory
RtOpenEvent
RtOpenMutex
RtOpenSemaphore
RtOpenSharedMemory
RtPulseEvent
RtReleaseMutex
RtReleaseSemaphore
RtResetEvent
RtSetEvent
RtWaitForMultipleObjects
RtWaitForSingleObject

Memory APIs

Memory Real-Time APIs	Memory Win32-Supported APIs
RtAllocateContiguousMemory	GetProcessHeap
RtAllocateLockedMemory	HeapAlloc
RtCommitLockHeap	HeapCreate
RtCommitLockProcessHeap	HeapDestroy
RtCommitLockStack	HeapFree
RtCreateSharedMemory	HeapReAlloc
RtFreeContiguousMemory	HeapSize
RtFreeLockedMemory	
RtGetPhysicalAddress	
RtLockKernel	
RtLockProcess	

Memory Real-Time APIs	Memory Win32-Supported APIs
RtMapMemory	
RtUnlockKernel	
RtUnlockProcess	
RtUnmapMemory	

Port and Bus IO APIs

Port and Bus IO Real-Time APIs
RtDisablePortIo
RtEnablePortIo
RtGetBusDataByOffset
RtReadPortBufferUchar, Ushort, Ulong
RtReadPortUchar, Ushort, Ulong
RtSetBusDataByOffset
RtTranslateBusAddress
RtWritePortBufferUchar, Ushort, Ulong
RtWritePortUchar, Ushort, Ulong

Processes and Threads APIs

Processes and Threads Real-Time APIs	Processes and Threads Win32-Supported APIs
RtCreateProcess	CreateThread
RtGetExitCodeProcess	ExitProcess
RtGetThreadTimeQuantum	ExitThread
RtIsInRtss	GetCurrentProcessId
RtTerminateProcess	GetCurrentThread
RtSetThreadPriority	GetCurrentThreadId
RtSetThreadTimeQuantum	GetExitCodeThread
RtSleepFt	GetThreadPriority
	RtOpenProcess
	ResumeThread
	SetThreadPriority
	Sleep
	SuspendThread
	TerminateThread
	TlsAlloc
	TlsFree
	TlsGetValue
	TlsSetValue

CHAPTER 2

Real-Time API

RtAllocateContiguousMemory

RtAllocateContiguousMemory allocates physically contiguous memory.

PVOID
RtAllocateContiguousMemory(
ULONG *Length*,
LARGE_INTEGER *PhysicalAddress*
);

Parameters

Length

An unsigned 32-bit quantity indicating the amount of memory, in bytes, to allocate.

PhysicalAddress

The highest physical address that can be part of the range allocated.

Return Values

The function returns a pointer to the memory allocated if successful; otherwise, it returns a **NULL** pointer.

Comments

RtAllocateContiguousMemory allocates memory in the virtual address space of the process, backed by contiguous, non-paged physical memory. The second argument allows the user to specify the highest acceptable physical address. If this memory is used with certain hardware devices, an upper-addressing limit may be imposed by the design of the hardware device.

Note that the amount of non-paged, contiguous memory available is relatively limited, and is rapidly fragmented through normal system operation. Applications should use this resource carefully and obtain allocations early in operation.

See Also

RtFreeContiguousMemory
RtGetPhysicalAddress

RtAllocateLockedMemory

RtAllocateLockedMemory commits and locks the specified amount of memory to avoid page faults as the memory is used.

PVOID
RtAllocateLockedMemory(
UINT *nNumberOfBytes*
);

Parameters

nNumberOfBytes

An unsigned integer specifying the number of bytes to allocate, commit, and lock.

Return Values

The function returns a pointer to the memory allocated if successful; otherwise, it returns a **NULL** pointer.

Comments

RtAllocateLockedMemory allocates memory in the virtual address space of the process, commits that space to physical memory, and locks that physical memory. The committed locked memory will not incur page faults when the memory is used, nor will the system page the allocated memory out to secondary storage.

See Also

RtCommitLockHeap
RtCommitLockProcessHeap
RtCommitLockStack
RtFreeLockedMemory
RtLockKernel
RtLockProcess
RtUnlockKernel
RtUnlockProcess

RtAtoi

RtAtoi converts a given string value to an integer.

```
INT
RtAtoi(
LPCSTR lpString
    );
```

Parameters

lpString

The source character string.

Return Values

This function returns the integer value of the string.

Comments

RtAtoi is similar to **atoi**, but **RtAtoi** does not require the C run-time library and can work with any combination of run-time libraries.

This function supports decimal digits only, and does not allow leading whitespace or signs.

See Also

RtPrintf
RtWprintf
RtWtoi

RtAttachInterruptVector

RtAttachInterruptVector allows the user to associate a handler routine in user space with a hardware interrupt. Non-shared, level-triggered interrupts are supported only in the RTSS environment. They are not supported in a Win32 environment.

Note: **RtAttachInterruptVector** does not permit shared interrupts. The function call, **RtAttachInterruptVectorEx**, permits shared interrupts.

HANDLE
RtAttachInterruptVector(
PSECURITY_ATTRIBUTES *pThreadAttributes*,
ULONG *StackSize*,
VOID (RTFCNDCL **pRoutineIST*)(PVOID *contextIST*),
PVOID *Context*,
ULONG *Priority*,
INTERFACE_TYPE *InterfaceType*,
ULONG *BusNumber*,
ULONG *BusInterruptLevel*,
ULONG *BusInterruptVector*
);

Parameters

pThreadAttributes (ignored by RTSS)

A security attributes structure used when the handler thread is created. See **CreateThread** in the Windows Win32 SDK.

StackSize

The number of bytes to allocate for the handler thread's stack. The stack size defaults to the thread's stack size.

pRoutineIST

A pointer to the handler routine to be run. The routine takes a single PVOID argument, and context and returns VOID.

ContextIST

The argument to the handler routine, cast as a PVOID.

Priority

The thread priority for the handler routine. Executing equal and higher priority threads disable the interrupt; executing lower priority threads enable it.

InterfaceType

The type of bus interface on which the hardware is located. It can be one of the following types: **Internal, ISA, EISA, MicroChannel, TurboChannel,** or **PCIBus**. The upper boundary on the bus types supported is always **MaximumInterfaceType**.

BusNumber

The bus that the device is on in a multiple bus environment. It is zero-based. Note that this value applies to each bus type, so for a system with one ISA bus and one PCIBus, for example, each would have a BusNumber of 0.

BusInterruptLevel

A bus-specific interrupt level associated with the routine that will handle the interrupt.

BusInterruptVector

A bus-specific address associated with the routine that will handle the interrupt.

Return Values

The function returns an RTX-specific interrupt handle if successful. The function returns a NULL handle for an invalid argument, for failing to connect the interrupt, or if a device is already using the bus resources requested.

Comments

RtAttachInterruptVector allows a user to associate a handling routine with a hardware interrupt on one of the supported buses on the computer. The routine uses the DDK HAL routines for getting the system-wide interrupt vector based on the bus-specific interrupt level and vector. If successful, it creates a handling thread in the user's application. When the interrupt occurs, the handling thread is notified and the thread runs the handling routine and argument specified.

The handling routine must not change its thread priority.

As in a typical device driver, the user is responsible for initializing the hardware device, enabling the generation of interrupts, and acknowledging interrupts in the appropriate manner. Interrupt generation on the device can be enabled after a successful call to RtAttachInterruptVector. Conversely, the user must disable interrupts before disconnecting the interrupt with **RtReleaseInterruptVector**, and also disconnect the interrupt before exiting. In the interrupt handling routine, the user should perform the appropriate steps to acknowledge the device's interrupt. Typically, these operations are performed by writing commands to a device's command/status register, which is either memory-mapped or in the I/O address space of the system.

See Also

HalGetInterruptVector (in the *DDK for Windows NT 4.0 and DDK for Windows 2000*)

RtEnablePortIo

RtReadPortBufferUchar

RtReleaseInterruptVector

RtWritePortUchar

RtAttachInterruptVectorEx

RtAttachInterruptVectorEx allows the user to associate an IST and ISR with a shared or non-shared hardware interrupt.

HANDLE
RtAttachInterruptVectorEx(
PSECURITY_ATTRIBUTES *pThreadAttributes*,
ULONG *StackSize*,
BOOLEAN (*RTFCNDCL *pRoutine*IST)(*PVOID ContextIST*),
PVOID *ContextIST*,
ULONG *Priority*,
INTERFACE_TYPE *InterfaceType*,
ULONG *BusNumber*,
ULONG *BusInterruptLevel*,
ULONG *BusInterruptVector*,
BOOLEAN *ShareVector*,
KINTERRUPT_MODE *InterruptMode*,
INTERRUPT_DISPOSITION (*RTFCNDCL *pRoutineISR*)(*PVOID ContextISR*)
);

Parameters

pThreadAttributes (Ignored by RTSS)

A security attributes structure used when the handler thread is created. See **CreateThread** in the *Windows Win32 SDK*.

StackSize

The number of bytes to allocate for the handler thread's stack. See **CreateThread**.

pRoutineIST

A pointer to the handler routine to be run by the interrupt service thread (IST). The routine takes a single PVOID argument and returns BOOLEAN.

Context

The argument to the handler routines (i.e., the routines pointed to by *pRoutine* and *MyInterrupt*), cast as a PVOID.

Priority

The thread priority for the handler routine. Executing equal and higher priority threads disable the interrupt; executing lower priority threads enable it.

InterfaceType

The type of bus interface on which the hardware is located. It can be one of the following types: **Internal**, **ISA**, **EISA**, **MicroChannel**, **TurboChannel**, or **PCIBus**. The upper boundary on the bus types supported is always **MaximumInterfaceType**.

BusNumber

The bus the device is on in a multiple bus environment, with counting starting at zero. Typically, a machine has only one or two of a particular bus type, so this value is either 0 or 1. Note that this value applies to each bus type, so for a system with one ISA bus and one PCI bus, for example, each would have a BusNumber of 0.

BusInterruptLevel

A bus-specific interrupt level associated with the device.

BusInterruptVector

A bus-specific interrupt vector associated with the device.

ShareVector

Specifies whether the caller is prepared to share the interrupt vector.

InterruptMode

Specifies whether the device interrupt is **LevelSensitive** or **Latched**.

pRoutineISR

A pointer to a routine to be run by the interrupt service routine (ISR) or NULL to specify no routine. The routine takes a single PVOID argument, ContextIST and returns INTERRUPT_DISPOSITION.

Return Values

The function returns an RTX-specific interrupt handle if successful. The function returns a NULL handle for an invalid argument, for failing to connect the interrupt, or if a device is already using the bus resources requested and either a previous attachment or this attachment specified ShareVector as FALSE.

Comments

RtAttachInterruptVectorEx allows a user to associate two handling routines with a hardware interrupt on one of the supported buses on the computer. It uses the DDK HAL routines for getting the system-wide interrupt vector based on the bus-specific interrupt level and vector. If successful, it creates a handling thread in the user's application.

When the interrupt occurs, the ISR calls the *MyInterrupt* routine (if present). This optional routine should be used to determine which attachment of a group of shared attachments should handle the interrupt. Since it is called at interrupt level, this routine should complete its work quickly; 1-2 msec is recommended. The routine should restrict itself to the calls in the Port I/O or mapped data transfer API (**RtReadPort. . .**, **RtWritePort. . .**). The routine pointed to by MyInterrupt returns one of three values.

- It returns **PassToNextDevice** if its associated device did not generate the interrupt.
- It returns **CallInterruptThread** if its associated device generated the interrupt and the handling thread should be notified and call the function pointed to by pRoutineISR to handle the interrupt.

- It returns **Dismiss** if its associated device generated the interrupt and no further action is required of the handling thread (presumably because this routine has done all that is necessary).

When they are called, the handling routines are passed *Context* as an argument.

The handling routine called via *pRoutine* **must not** change its thread priority.

As in a typical device driver, the user is responsible for initializing the hardware device, enabling the generation of interrupts, and acknowledging interrupts in the appropriate manner. Interrupt generation on the device can be enabled after a successful call to **RtAttachInterruptVectorEx**. Conversely, the user must disable interrupts before disconnecting the interrupt with **RtReleaseInterruptVector**, and the user must disconnect the interrupt before exiting. In the interrupt handling routine, the user should perform the appropriate steps to acknowledge the device's interrupt. Typically, these operations are performed by writing commands to a device's command/status register, which is either memory-mapped or in the I/O address space of the system.

If *ShareVector* is specified as true, the interrupt handling routine must be prepared to share the specified interrupt vector with other devices. When an interrupt occurs and the handling routine is run, it should check the device in an appropriate manner to determine if the interrupt came from its device. If so, it handles and acknowledges the interrupt in the usual way, and returns TRUE. If its device did not generate the interrupt, the handling routine returns FALSE.

If the vector is already in use by a Windows NT device driver, or another RTX program that did not specify *ShareVector* as TRUE, **RtAttachInterruptVectorEx** will fail.

See Also

HalGetInterruptVector (in the DDK for Windows NT 4.0 and DDK for Windows 2000)
RtAttachInterruptVector
RtEnablePortIo
RtReadPortUchar
RtReleaseInterruptVector
RtWritePortUchar

RtAttachShutdownHandler

RtAttachShutdownHandler registers a stop notification handler function with RTSS. The handler function is called in its own thread when one of the system stop events occurs.

HANDLE
RtAttachShutdownHandler(
PSECURITY_ATTRIBUTES pThreadAttributes,
ULONG Stacksize,
VOID (RTFCNDCL *Routine) (PVOID Context, LONG reason),
PVOID Context,
ULONG Priority
);

Parameters

*pThreadAttributes (*unused*)*

A security attributes structure used when the handler thread is created.

StackSize

The number of bytes to allocate for the handler thread's stack.

Routine

The handler function to call when RTSS delivers the stop notification.

Context

The argument to the handler routine.

Priority

The priority for the created thread.

Return Values

A stop handler object has been correctly instantiated when a valid handle is returned. Otherwise, INVALID_HANDLE_VALUE is returned and **GetLastError** should be called for more detailed information.

Comments

The function pointed to by *Routine* is called when Windows NT and Windows 2000 shut down. The source of the stop notification is presented to this function in the *reason* argument. The *reason* argument may have one of the following values according the reason for the notification:

RT_SHUTDOWN_NT_SYSTEM_SHUTDOWN	The system is starting a normal shutdown. Shortly after all shutdown handlers have been executed, Windows NT and Windows will stop.
RT_SHUTDOWN_NT_STOP	Windows NT or Windows 2000 have stopped (i.e., blue screen or stop screen). RTSS will continue to operate with service restrictions.

The order in which the shutdown handlers execute is dependent upon the priority specified when the handler object was created. This priority is simply the RTSS thread priority.

Only one shutdown handler object is permitted per RTSS process. A handler function should not call **ExitThread**, but should simply return when finished. When all registered shutdown handlers have returned, the system completes the shutdown sequence.

A shutdown handler object may be destroyed by calling **RtReleaseShutdownHandler**.

Note: There are a limited number of calls that can be made.

See Also

RtReleaseShutdownHandler

RtCancelTimer

RtCancelTimer cancels the expiration of the indicated timer.

```
BOOL
RtCancelTimer(
    HANDLE hTimer,
    PLARGE_INTEGER pTimeRemaining
    );
```

Parameters

hTimer

An RTX-specific handle to the timer.

pTimeRemaining

A pointer to a LARGE_INTEGER to store the time remaining on the canceled timer. If the pointer is non-NULL, the LARGE_INTEGER will be written with the time remaining on the timer at the time of cancellation. The time remaining is calculated relative to the current value of of the clock associated with the time at creation and is specified in 100ns units.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtCancelTimer cancels the specified timer, but does not delete it. If the user provides a non-NULL pointer to a LARGE_INTEGER, then time remaining on the timer at the time of cancellation is returned.

See Also

RtCreateTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTime
RtGetClockTimerPeriod
RtSetClockTime
RtSetTimer
RtSetTimerRelative
RtSleepFt
Sleep

RtCloseHandle

RtCloseHandle closes an open object handle.

BOOL
RtCloseHandle(
HANDLE *hObject*
);

Parameters

hObject

An open object handle.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

RtCloseHandle closes handles to the following RTSS objects:

- Event
- Interrupts
- Mutex
- Semaphore
- Shared memory
- Shutdown handler
- Timers

RtCloseHandle invalidates the specified object handle, decrements the object's handle count, and performs object retention checks. Once the last handle to an object is closed, the object is removed from the operating system.

Note: Threads must be closed with **CloseHandle**.

See Also

CloseHandle

RtCommitLockHeap

RtCommitLockHeap commits and locks the heap to avoid page faults as the heap is used.

```
BOOL
RtCommitLockHeap(
    HANDLE hHeap,
    ULONG nNumberOfBytes,
    VOID (RTFCNDCL *pExceptionRoutine)(HANDLE),
    );
```

Parameters

hHeap

A handle to the heap to be committed and locked.

nNumberOfBytes

The number of bytes in the heap to lock.

*pExceptionRoutine (*ignored*)*

The exception routine to call if the heap uses more than the locked amount.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtCommitLockHeap commits and locks the specified heap in physical memory so that it does not incur page faults as the memory is used, and the memory used for the heap is not paged out by the system.

Since all RTSS heaps are always locked, this function has no effect in the RTSS environment.

See Also

RtAllocateLockedMemory
RtCommitLockProcessHeap
RtCommitLockStack
RtFreeLockedMemory
RtLockKernel
RtLockProcess
RtUnlockKernel
RtUnlockProcess

RtCommitLockProcessHeap

RtCommitLockProcessHeap commits and locks the system process heap to avoid page faults as the heap is used.

```
BOOL
RtCommitLockProcessHeap(
    ULONG nNumberOfBytes,
    VOID (RTFCNDCL *pExceptionRoutine)(HANDLE),
    );
```

Parameters

nNumberOfBytes

The number of bytes in the heap to lock.

**ExceptionRoutine (*ignored*)*

The exception routine to call if the heap uses more than the locked amount.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtCommitLockProcessHeap commits and locks the system process heap in physical memory so that it does not incur page faults as the memory is used, and the memory used for the heap is not paged out by the system.

Since all RTSS heaps are always locked, this function has no effect in the RTSS environment.

See Also

RtAllocateLockedMemory
RtCommitLockHeap
RtCommitLockStack
RtFreeLockedMemory
RtLockKernel
RtLockProcess
RtUnlockKernel
RtUnlockProcess

RtCommitLockStack

RtCommitLockStack commits and locks the specified stack to avoid page faults as the stack is used.

```
BOOL
RtCommitLockStack(
    ULONG nNumberOfBytes
    );
```

Parameters

nNumberOfBytes

The number of bytes in the stack to lock.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtCommitLockStack commits and locks the stack in physical memory so that it does not incur page faults as the memory is used, and the memory used for the stack is not paged out by the system.

The RTSS stack is always locked. Any attempt to lock the stack beyond the stack size at thread creation fails; otherwise, this call has no effect in the RTSS environment.

See Also

RtAllocateLockedMemory
RtCommitLockHeap
RtCommitLockProcessHeap
RtFreeLockedMemory
RtLockKernel
RtLockProcess
RtUnlockKernel
RtUnlockProcess

RtCreateEvent

RtCreateEvent creates a named or unnamed event object.

HANDLE
RtCreateEvent(
LPSECURITY_ATTRIBUTES *lpEventAttributes*,
BOOL *ManualReset*,
BOOL *pInitialState*,
LPCTSTR *lpName*
);

Parameters

lpEventAttributes (ignored)

A pointer to a SECURITY_ATTRIBUTES structure.

bManualReset

Specifies whether a manual-reset or auto-reset event is created. If TRUE, then use the **RtResetEvent** function to manually reset the state to non-signaled. If FALSE, the system automatically resets the state to non-signaled after a single waiting thread has been released.

bInitialState

The initial state of the event object. If TRUE, the initial state is signaled; otherwise, it is non-signaled.

lpName

A pointer to a null-terminated string specifying the name of the event object. The name is limited to RTX_MAX_PATH characters, and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

If *lpName* matches the name of an existing named event object, this function requests access to the existing object. In this case, *bManualReset* and *bInitialState* are ignored because they have already been set by the creating process.

If *lpName* matches the name of an existing mutex, semaphore, or shared memory object, the function fails and **GetLastError** returns ERROR_INVALID_HANDLE. This occurs because event, mutex, semaphore, and shared memory objects share the same namespace.

If *lpName* is NULL, the event object is created without a name.

Return Values

If the function succeeds, the return value is a handle to the event object. If the named event object existed before the function call, **GetLastError** returns ERROR_ALREADY_EXISTS. Otherwise, **GetLastError** returns zero.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

The handle returned by **RtCreateEvent** has all accesses to the new event object and can be used in any function that requires a handle to an event object.

Any thread of the calling process can specify the event-object handle in a call to one of the wait functions. The wait functions return when the state of the specified object is signaled. When a wait function returns, the waiting thread is released to continue its execution.

The initial state of the event object is specified by the *bInitialState* parameter. Use the **RtSetEvent** function to set the state of an event object to signaled. Use the **RtResetEvent** function to reset the state of an event object to non-signaled.

When the state of a manual-reset event is signaled, it remains signaled until it is explicitly reset to non-signaled by the **RtResetEvent** function. Any number of waiting threads, or threads that subsequently begin wait operations for the specified event object, can be released while the object's state is signaled.

When the state of an auto-reset event object is signaled, it remains signaled until a single waiting thread is released; the system then resets the state to non-signaled. If no threads are waiting, the event object remains signaled.

Multiple processes can have handles of the same event object, enabling use of the object for inter-process synchronization. The available object-sharing mechanism is: A process can specify the name of a event object in a call to **RtOpenEvent** or **RtCreateEvent**.

Use **RtCloseHandle** to close the handle. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

See Also

RtCloseHandle
RtOpenEvent
RtPulseEvent
RtResetEvent
RtSetEvent

RtCreateMutex

RtCreateMutex creates an RTSS mutex. A handle is returned to the newly created mutex object.

HANDLE
RtCreateMutex(
LPSECURITY_ATTRIBUTES *lpMutexAttributes*,
BOOL *bInitialOwner*,
LPCTSTR *lpName*
);

Parameters

*lpMutexAttributes (*ignored*)*

A pointer to a SECURITY_ATTRIBUTES structure.

bInitialOwner

The initial ownership state of the mutex object. If this value is TRUE and the caller created the mutex, the calling thread obtains ownership of the mutex object. Otherwise, the calling thread does not obtain ownership of the mutex.

lpName

A pointer to a null-terminated string specifying the name of the mutex object. The name is limited to RTX_MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

If *lpName* matches the name of an existing named mutex object, this function requests MUTEX_ALL_ACCESS access to the existing object. In this case, the *bInitialOwner* parameter is ignored because it has already been set by the creating process.

If *lpName* matches the name of an existing event, semaphore, or shared memory object, the function fails and **GetLastError** returns ERROR_INVALID_HANDLE. This occurs because event, mutex, semaphore, and shared memory objects share the same namespace.

If *lpName* is NULL, the mutex object is created without a name.

Return Values

If the function succeeds, the return value is a handle to the mutex object. If the named mutex object existed before the function call, **GetLastError** returns ERROR_ALREADY_EXISTS. Otherwise, **GetLastError** returns zero.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

The handle returned by **RtCreateMutex** has MUTEX_ALL_ACCESS access to the new mutex object and can be used in any function that requires a handle to a mutex object.

Any thread of the calling process can specify the mutex-object handle in a call to any wait function.

The state of a mutex object is signaled when it is not owned by any thread. The creating thread can use the *bInitialOwner* flag to request immediate ownership of the mutex. Otherwise, a thread must use the wait function to request ownership. When the mutex's state is signaled, the highest priority waiting thread is granted ownership (if more than one thread is waiting at the same priority, they receive ownership of the mutex in the order they waited); the mutex's state changes to non-signaled; and the wait function returns. Only one thread can own a mutex at any given time. The owning thread uses **RtReleaseMutex** to release its ownership.

The thread that owns a mutex can specify the same mutex in repeated wait function calls without blocking its execution. Typically, you would not wait repeatedly for the same mutex, but this mechanism prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call **RtReleaseMutex** once for each time that the mutex satisfied a wait.

Two or more processes can call **RtCreateMutex** to create the same named mutex. The first process actually creates the mutex, and subsequent processes open a handle to the existing mutex. This enables multiple processes to get handles of the same mutex, while relieving the user of the responsibility of ensuring that the creating process is started first. When using this technique, set the *bInitialOwner* flag to FALSE; otherwise, it can be difficult to be certain which process has initial ownership.

Multiple processes can have handles of the same mutex object, enabling use of the object for process synchronization. The available object-sharing mechanism is: A process can specify the name of a mutex object in a call to **RtOpenMutex** or **RtCreateMutex**.

RtCloseHandle closes a mutex-object handle. The system closes the handle automatically when the process terminates. The mutex object is destroyed when its last handle has been closed.

See Also

RtCloseHandle
RtOpenMutex
RtReleaseMutex

RtCreateProcess

RtCreateProcess creates and starts a new RTSS process. The new RTSS process runs the specified RTSS executable file. RtCreateProcess is supported only in the Win32 environment.

BOOL
RtCreateProcess(
LPCTSTR lpApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation
);

Parameters

lpApplicationName

Pointer to a null-terminated string that specifies the module to execute. The string must specify the full path and file name of the module to execute.

The *lpApplicationName* parameter can be NULL. In that case, the module name must be the first white space-delimited token in the *lpCommandLine* string.

When *lpCommandLine* is non-NULL and you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin. For example, **`"C:\Program Files\Rtx Test.rtss" argv1 argv2`** is a valid string for this parameter. When *lpCommandLine* is NULL, quotes are not needed.

The specified module must be an RTSS application.

lpCommandLine

Pointer to a null-terminated string that specifies the command line to execute. The system adds a null character to the command line.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-NULL, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use **GetCommandLine** to retrieve the entire command line. C runtime processes can use the **argc** and **argv** arguments.

If *lpApplicationName* is NULL, the first white-space - delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the

explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .rtss is appended.

lpProcessAttributes (ignored)

lpThreadAttributes (ignored)

bInheritHandles (ignored)

dwCreationFlags (ignored)

lpEnvironment (ignored)

lpCurrentDirectory (ignored)

lpStartupInfo (ignored)

lpProcessInformation

Pointer to a PROCESS_INFORMATION structure that receives identification information about the new process.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Comments

An RTSS process can be created and started through **RTSSrun** in command line windows. As an alternative way, **RtCreateProcess** can be used to create and start a new RTSS process through Win32 programs. **RtCreateProcess** uses **RTSSrun** (without switches) to create and start the new RTSS process.

When created, the new RTSS process handle receives full access rights. The handle can be used in any function that requires an object handle to that type. The RTSS process handle is returned in the PROCESS_INFORMATION structure.

The RTSS process is assigned a process identifier. The identifier is valid until the RTSS process terminates. It can be used to identify the RTSS process, or specified in **RtOpenProcess** to open a handle to the RTSS process. The RTSS process identifier is returned in the PROCESS_INFORMATION structure.

RtCreateProcess does not create a handle for the primary thread of the new RTSS process. The returning value of *lpProcessInformation-hThread* is NULL and *lpProcessInformation-dwThreadId* is 0.

The preferred way to shut down an RTSS process is by using **ExitProcess**, because this function sends notification of approaching termination to all RTDLLs attached to the RTSS process. Other means of shutting down an RTSS process do not notify the attached RTDLLs.

The created RTSS process remains in the system until all threads within the RTSS process have terminated and all handles to the RTSS process and any of its threads have been closed through calls to **CloseHandle**. The handle for the RTSS process must be closed through a call to **CloseHandle**.

If this handle is not needed, it is best to close it immediately after the process is created. When the last thread in an RTSS process terminates, the following events occur:

- All objects opened by the RTSS process are implicitly closed.
- The process's termination status (which is returned by **RtGetExitCodeProcess**) changes from its initial value of STILL_ACTIVE to the termination status of the last thread to terminate.
- The RTSS process object is set to the signaled state, satisfying any threads that were waiting on the object.

See Also

CloseHandle
ExitProcess
RtGetExitCodeProcess
RtOpenProcess
RtTerminateProcess

RtCreateSemaphore

RtCreateSemaphore creates a named or unnamed semaphore object.

HANDLE
RtCreateSemaphore(
LPSECURITY_ATTRIBUTES *lpSemaphoreAttributes*,
LONG *lInitialCount*,
LONG *lMaximumCount*,
LPCTSTR *lpName*
);

Parameters

*lpSemaphoreAttributes (*ignored*)*

A pointer to a SECURITY_ATTRIBUTES structure.

lInitialCount

An initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to *lMaximumCount*. The state of a semaphore is signaled when its count is greater than zero and non-signaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling **RtReleaseSemaphore**.

lMaximumCount

The maximum count for the semaphore object. This value must be greater than zero.

lpName

A pointer to a null-terminated string specifying the name of the semaphore object. The name is limited to RTX_MAX_PATH characters, and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

If *lpName* matches the name of an existing named semaphore object, this function requests access to the existing object. In this case, *lInitialCount* and *lMaximumCount* are ignored because they have already been set by the creating process.

If *lpName* matches the name of an existing event, mutex, or shared memory object, the function fails and **GetLastError** returns ERROR_INVALID_HANDLE. This occurs because event, mutex, semaphore, and shared memory objects share the same namespace.

If *lpName* is NULL, the semaphore object is created without a name.

Return Values

If the function succeeds, the return value is a handle to the semaphore object. If the named semaphore object existed before the function call, GetLastError returns ERROR_ALREADY_EXISTS. Otherwise, **GetLastError** returns zero.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

The handle returned by **RtCreateSemaphore** has all accesses to the new semaphore object and can be used in any function that requires a handle to a semaphore object.

Any thread of the calling process can specify the semaphore-object handle in a call to one of the wait functions. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any object is signaled. When a wait function returns, the waiting thread is released to continue its execution.

The state of a semaphore object is signaled when its count is greater than zero, and non-signaled when its count is equal to zero. *lInitialCount* specifies the initial count. Each time a waiting thread is released because of the semaphore's signaled state, the count of the semaphore is decreased by one. Use **RtReleaseSemaphore** to increment a semaphore's count by a specified amount. The count can never be less than zero or greater than the value specified in *lMaximumCount*.

Multiple processes can have handles of the same semaphore object, enabling use of the object for inter-process synchronization. The available object-sharing mechanism is a process that can specify the name of a semaphore object in a call to **RtOpenSemaphore** or **RtCreateSemaphore**.

Use **RtCloseHandle** to close the handle. The system closes the handle automatically when the process terminates. The semaphore object is destroyed when its last handle has been closed.

See Also

RtCloseHandle
RtOpenSemaphore
RtReleaseSemaphore

RtCreateSharedMemory

RtCreateSharedMemory creates a named region of physical memory that can be mapped by any process.

HANDLE
RtCreateSharedMemory(
DWORD *flProtect*,
DWORD *MaximumSizeHigh*,
DWORD *MaximumSizeLow*,
LPCTSTR *lpName*,
VOID ** *location*
);

Parameters

flProtect (ignored by RTSS)

The protection desired for the shared memory view. This parameter can be one of the following values:

PAGE_READONLY
Gives read-only access to the committed region of pages. An attempt to write to or execute the committed region results in an access violation.

PAGE_READWRITE
Gives read-write access to the committed region of pages.

MaximumSizeHigh

The high-order 32 bits of the size of the shared memory object.

MaximumSizeLow

The low-order 32 bits of the size of the shared memory object.

lpName

A pointer to a null-terminated string specifying the name of the shared memory object. The name can contain any character except the backslash (\).

If this parameter matches the name of an existing named shared memory object, the function requests access to the shared memory object with the protection specified by *flProtect*.

If lpName matches the name of an existing event, mutex, or semaphore object, the function fails and **GetLastError** returns ERROR_INVALID_HANDLE. This occurs because event, mutex, semaphore, and shared memory objects share the same namespace.

If *lpName* is NULL, the mapping object is created without a name.

location

A pointer to a location where the virtual address of the shared memory will be stored.

Return Values

If the function succeeds, the return value is a handle to the shared memory object. If the object existed before the function call, **GetLastError** returns ERROR_ALREADY_EXISTS, and the return value is a valid handle to the existing shared memory object (with its current size, not the new specified size). If the mapping object did not exist, **GetLastError** returns zero and the location is set.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

The handle that **RtCreateSharedMemory** returns has full access to the new shared memory object. Shared memory objects can be shared by name. For information on opening a shared memory object by name, see **RtOpenSharedMemory**.

To close a shared memory object, an application must close its handle by calling **RtCloseHandle**.

When all handles to the shared memory object representing the physical memory are closed, the object is destroyed and physical memory is returned to the system.

See Also

RtCloseHandle
RtOpenSharedMemory

RtCreateTimer

RtCreateTimer creates a timer associated with the specified clock, and returns a handle to the timer.

```
HANDLE
RtCreateTimer(
    PSECURITY_ATTRIBUTES pThreadAttributes,
    ULONG StackSize,
    VOID (RTFCNDCL *Routine) (PVOID context),
    PVOID Context,
    ULONG Priority,
    CLOCK Clock
    );
```

Parameters

pThreadAttributes (ignored by RTSS)

An optional pointer to a SECURITY_ATTRIBUTES structure to be used at handler thread creation. Pass in NULL for default.

StackSize

The stack size for handler thread. Use a size of 0 for default.

Routine

A pointer to the routine to be run upon completion. The routine takes a single PVOID argument and returns VOID.

Context

The argument to the routine, cast as a PVOID.

Priority

The handler thread priority as defined below.

Clock

A clock identifier as defined below in the Comments sections.

Return Values

If successful, the function returns a non-zero handle to the timer; otherwise, it returns a NULL handle. To set the timer to expire, see **RtSetTimer** or **RtSetTimerRelative**. Upon expiration, the specified routine is run with the specified argument.

Comments

RtCreateTimer allocates a new timer and returns a handle to it. Legal clock values, as enumerated in rtapi.h, are listed below:

Clock Value	Meaning
CLOCK_1	One millisecond timer.
CLOCK_2	Real-time HAL timer. Default is 500 microseconds (as specified in the registry).
CLOCK_FASTEST	The fastest available clock and time on the system. This is usually CLOCK_2.
CLOCK_SYSTEM	Same as CLOCK_1.

The timer routine will run as a separate handling thread. *pThreadAttributes*, *StackSize*, and *Priority* are used to control the creation of the handler thread. See **CreateThread** and **SetThreadPriority** for details on these parameters.

To run a different handling routine/context, a new timer must be created.

See Also

RtCancelTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTime
RtGetClockTimerPeriod
RtSetClockTime
RtSetTimer
RtSetTimerRelative

RtDeleteTimer

RtDeleteTimer deletes the timer specified by the given handle.

```
BOOL
RtDeleteTimer(
    HANDLE hTimer
    );
```

Parameters

hTimer

An RTX-specific handle to the timer to be deleted.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if an invalid parameter is specified.

Comments

RtDeleteTimer deletes the specified timer, first canceling it if it has been scheduled to expire. Note that timer handles are not Windows NT object handles, and the RTX timer subsystem does not maintain a reference count. Deleting a timer removes the timer entirely. **RtCloseHandle** can also be used to delete a timer.

See Also

RtCancelTimer
RtCloseHandle
RtCreateTimer
RtGetClockResolution
RtGetClockTime
RtGetClockTimerPeriod
RtSetClockTime
RtSetTimer
RtSetTimerRelative

RtDisableInterrupts

In the Win32 environment, **RtDisableInterrupts** disables all user-level interrupt handling for all interrupts to which the Win32 process is attached.

In an RTSS environment, **RtDisableInterrupts** disables all interrupts at the processor level including timer interrupts.

BOOL
RtDisableInterrupts(VOID)

Return Values

The function returns TRUE if successful; otherwise it returns FALSE.

Comments

To minimize latencies for higher priority threads, **RtEnableInterrupts** should be called as soon as possible after **RtDisableInterrupts**.

For Win32 processes, this function does not program the hardware to stop generating interrupts. Such programming must be done separately, typically via port I/O calls to the command/status registers for the device.

See Also

RtAttachInterruptVector
RtEnableInterrupts
RtEnablePortIo
RtReadPortUchar
RtReleaseInterruptVector
RtWritePortUchar

RtDisablePortIo

RtDisablePortIo disables direct I/O port access from user context.

BOOL
RtDisablePortIo(
PUCHAR *StartPort*,
ULONG *nNumberOfBytes*
);

Parameters

StartPort

The first port to have direct I/O permissions disabled by this call. Each I/O space address points at a single byte. For ports that represent 2-byte or 4-byte locations, the appropriate number of I/O space addresses (two and four, respectively) must be disabled.

nNumberOfBytes

An unsigned 32-bit integer indicating the number of addresses/bytes to disable, starting at *StartPort*.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtDisablePortIo disables direct port I/O access for ports from the user context.

This function currently has no impact on RTSS determinism. This call is a NO-OP (no operation) when issued from RTSS applications.

See Also

RtEnablePortIo
RtReadPortBuffer*(Uchar, Ushort, Ulong)
RtReadPort* (Uchar, Ushort, Ulong)
RtWritePortBuffer* (Uchar, Ushort, Ulong)
RtWritePort* (Uchar, Ushort, Ulong)

RtEnableInterrupts

RtEnableInterrupts enables user-level interrupt handling for all interrupts to which the process is attached.

VOID
RtEnableInterrupts(VOID);

Parameters

This function has no parameters.

Comments

In the Win32 environment, interrupt handling is automatically enabled after attaching to an interrupt successfully. Note that this function does not program the hardware to enable or generate interrupts. Such programming must be done separately, typically via port I/O calls to the command/status registers for the device.

See Also

HalGetInterruptVector (in the DDK for Windows NT and DDK for Windows 2000)
RtAttachInterruptVector
RtDisableInterrupts
RtDisablePortIo
RtReadPortUchar
RtReleaseInterruptVector
RtWritePortUchar

RtEnablePortIo

RtEnablePortIo enables direct I/O port access from user context.

BOOL
RtEnablePortIo(
PUCHAR *StartPort*,
ULONG *nNumberOfBytes*
);

Parameters

StartPort

The first port to have direct I/O permissions enabled by this call. Each I/O space address points at a single byte. For ports that represent 2-byte or 4-byte locations, the appropriate number of I/O space addresses must be enabled, or an exception will be encountered.

nNumberOfBytes

An unsigned 32-bit integer indicating the number of addresses/bytes to enable, starting at *StartPort*.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

This function currently has no impact on RTSS determinism. This call is a NO-OP (no operation) when issued from RTSS applications.

RtEnablePortIo enables direct user access to the specified range of I/O addresses. On the Pentium processor, there are 216-1 = 65,535 byte-wide I/O port addresses. Two-byte word and four-byte double word ports take two and four I/O port addresses, respectively, and fall on even word and long word addresses (i.e., divisible by two and four), respectively.

The address for which direct I/O is to be enabled is passed in StartPort, cast as a pointer to an unsigned character (i.e., a single-byte quantum). Generally, this address represents a hardware register or port, so only a single byte needs to be enabled (i.e., *nNumberOfBytes* is set to 1). If the location represents a two-byte word or four-byte double word port, then the parameter *nNumberOfBytes* should reflect the width of the I/O port. Note that *nNumberOfBytes* can also be used to enable permission for an entire range of addresses, independent of their data widths.

See Also

RtDisablePortIo
RtReadPortBuffer* (Uchar, Ushort, Ulong), RtReadPort* (Uchar, Ushort, Ulong)
RtWritePortBuffer* (Uchar, Ushort, Ulong), RtWritePort* (Uchar, Ushort, Ulong)

RtGetExitCodeProcess

RtGetExitCodeProcess retrieves the termination status of the specified process.

```
BOOL
RtGetExitCodeProcess(
    HANDLE hProcess,
    LPDWORD lpExitCode
    );
```

Parameters

hProcess

A handle to the process.

The handle must have PROCESS_QUERY_INFORMATION access.

lpExitCode

A pointer to a 32-bit variable to receive the process termination status.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

If the specified process has not terminated, the termination status returned is STILL_ACTIVE. If the process has terminated, the termination status returned may be one of the following values:

- The exit value specified in **ExitProcess** or **RtTerminateProcess**.
- The return value from **main** or **WinMain** of the process.

The exception value for an unhandled exception that caused the process to terminate.

RtFreeContiguousMemory

RtFreeContiguousMemory frees a previously allocated physically contiguous memory region.

BOOL
RtFreeContiguousMemory(
PVOID *pVirtualAddress*
);

Parameters

pVirtualAddress

A virtual address as returned by a previous call to RtAllocateContiguousMemory.

Return Values

The function returns TRUE if successful; otherwise, it returns FALSE.

Comments

RtFreeContiguousMemory releases a previous allocation of physically contiguous memory.

See Also

RtAllocateContiguousMemory
RtGetPhysicalAddress

RtFreeLockedMemory

RtFreeLockedMemory frees memory previously committed and locked by a call to RtAllocateLockedMemory.

BOOL
RtFreeLockedMemory(
PVOID *pVirtualAddress*
);

Parameters

pVirtualAddress

A pointer to the start of the memory, as returned by **RtAllocateLockedMemory**.

Return Values

The function returns TRUE if successful, otherwise, it returns FALSE.

Comments

RtFreeLockedMemory frees memory previously allocated, committed, and locked by **RtAllocateLockedMemory**.

See Also

RtAllocateLockedMemory
RtCommitLockHeap
RtCommitLockProcessHeap
RtCommitLockStack
RtLockKernel
RtLockProcess
RtUnlockKernel
RtUnlockProcess

RtGetBusDataByOffset

RtGetBusDataByOffset obtains details, starting at the given offset, about a given slot on an I/O bus.

ULONG
RtGetBusDataByOffset(
BUS_DATA_TYPE *BusDataType*,
ULONG *BusNumber*,
ULONG *SlotNumber*,
PVOID *pBuffer*,
ULONG *Offset*,
ULONG *Length*
);

Parameters

BusDataType

Type of bus data to be retrieved. Currently, its value can be **EisaConfiguration**, **Pos**, or **PCIConfiguration**. The upper bound on the bus types supported is always *MaximumBusDataType*.

BusNumber

Zero-based and system-assigned number of the bus in systems with more than one bus of the same *BusDataType*.

SlotNumber

Logical slot number. When PCIConfiguration is specified, this is a PCI_SLOT_NUMBER_TYPE value.

pBuffer

A pointer to a caller-supplied buffer for configuration information specific to *BusDataType*.

If EisaConfiguration is specified, the buffer will contain the CM_EISA_SLOT_INFORMATION structure followed by zero or more CM_EISA_FUNCTION_INFORMATION structures for the specified slot.

If Pos is specified, the buffer will contain a CM_MCA_POS_DATA structure for the specified slot.

When PCIConfiguration is specified, the buffer will contain some or all of the PCI_COMMON_CONFIG information for the given SlotNumber. The specified Offset and Length determine how

Offset

If the *BusDataType* is EisaConfiguration or Pos, the offset is zero. Otherwise, specifies the byte offset in the PCI_COMMON_CONFIG structure for which the requested information should be returned. Callers can use the system-defined constant,

PCI_COMMON_HDR_LENGTH, to specify the device-specific area of PCI_COMMON_CONFIG.

Length

Maximum number of bytes to return in the buffer.

Return Values

RtGetBusDataByOffset returns the number of bytes of data it wrote in the given buffer. If the given *BusDataType* is not valid for the current platform, this routine returns zero.

When the input BusType is PCIConfiguration, **RtGetBusDataByOffset** can return either of the following values to indicate an error:

Value	Meaning
0 (zero)	The specified PCIBus does not exist.
2	The specified PCIBus exists, but there is no device at the given PCI SlotNumber. The buffer also contains the value PCI_INVALID_VENDOR_ID at the PCI_COMMON_CONFIG VENDORID member.

Comments

This call can be used to locate devices on a particular I/O bus in the machine. The bus-type-specific configuration data returned can later be used in other calls, such as **RtSetBusDataByOffset** and **RtTranslateBusAddress**.

When accessing the device-specific area of the PCI configuration space, **RtGetBusDataByOffset** guarantees the following:

- This routine never reads or writes data outside the range specified by the input Offset and Length.
- Even if the input Length is exactly a byte or a (two-byte) word, this routine never accesses any data outside the requested range.

The PCI structures and values above are defined in rtapi.h. For more information, consult the *Device Driver Kit (DDK) for Windows NT and Windows 2000*.

See Also

HalGetBusDataByOffset (in the DDK for Windows NT and DDK for Windows 2000)

RtSetBusDataByOffset

RtTranslateBusAddress

RtGetClockResolution

RtGetClockResolution obtains the resolution of the specified clock.

```
BOOL
RtGetClockResolution(
    CLOCK Clock,
    PLARGE_INTEGER pResolution
    );
```

Parameters

Clock

A clock identifier.

pResolution

A pointer to a LARGE_INTEGER structure in which to store the results.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtGetClockResolution obtains the resolution of the specified clock. The resolution is specified in 100ns units. See the table in the **RtCreateTimer** Comments section for a list of legal clock values.

See Also

RtCancelTimer
RtCreateTimer
RtDeleteTimer
RtGetClockTime
RtGetClockTimerPeriod
RtSetClockTime
RtSetTimer
RtSetTimerRelative

RtGetClockTime

RtGetClockTime obtains the current value of the specified clock.

```
BOOL
RtGetClockTime(
    CLOCK Clock,
    PLARGE_INTEGER pTime
    );
```

Parameters

Clock

A clock identifier.

pTime

A pointer to a LARGE_INTEGER structure in which to store the results.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtGetClockTime obtains the value of the specified clock. The time is specified in 100ns units. See the table in the **RtCreateTimer** Comments section for a list of legal clock values.

See Also

RtCancelTimer
RtCreateTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTimerPeriod
RtSetClockTime
RtSetTimer
RtSetTimerRelative

RtGetClockTimerPeriod

RtGetClockTimerPeriod obtains the minimum timer period of the specified clock. The **RtGetClockTime** call delivers the clock time as 64-bit quantity of 100ns.

BOOL
RtGetClockTimerPeriod(
CLOCK *Clock*,
PLARGE_INTEGER *pTime*
);

Parameters

Clock

A clock identifier.

pTime

A pointer to a LARGE_INTEGER structure in which to store the results.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtGetClockTimerPeriod obtains the minimum timer period of the specified clock. Timers with an expiration interval smaller than this will produce unpredictable results. See the table in the **RtCreateTimer** Comments section for a list of legal clock values.

See Also

RtCancelTimer
RtCreateTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTime
RtSetClockTime
RtSetTimer
RtSetTimerRelative

RtGetPhysicalAddress

RtGetPhysicalAddress returns the physical address for the virtual address of a contiguous physical memory buffer previously allocated by **RtAllocateContiguousMemory**.

In the RTSS environment, this function applies to all memory allocation. In the Win32 environment, it applies to contiguous memory allocation only.

LARGE_INTEGER
RtGetPhysicalAddress(
PVOID *pVirtualAddress*
);

Parameters

pVirtualAddress

The virtual address of the base of a contiguous memory buffer as returned by **RtAllocateContiguousMemory**.

Return Values

If successful, the call returns the physical address corresponding to the base virtual address of the contiguous buffer. Otherwise, the call returns a NULL pointer, usually as a result of an invalid parameter.

Comments

RtGetPhysicalAddress allows the user to get the base physical address for a contiguous memory buffer allocated by **RtAllocateContiguousMemory**. The system physical addresses are eight-byte data types, returned as a LARGE_INTEGER.

In the RTSS environment, this call can be used on any virtual address.

See Also

RtAllocateContiguousMemory
RtFreeContiguousMemory

RtGetThreadPriority

RtGetThreadPriority returns the priority value for the specified thread.

RTSS Environment: RTSS has no distinct priority classes and the priority value specified is the only determination of a thread's priority.

Win32 Environment: The priority value, together with the priority class of the thread's process, determines the thread's base-priority level. All processes that attach to the Win32 RTAPI library are placed in the Win32 real-time priority class after a call to RtGetThreadPriority or RtSetThreadPriority.

INT
RtGetThreadPriority(
HANDLE *hThread*
);

Parameters

hThread

The thread identifier.

Return Values

If the function succeeds, the return value is the thread's priority level.

If the function fails, the return value is THREAD_PRIORITY_ERROR_RETURN. To get extended error information, call **GetLastError**.

Comments

Win32 threads are initially set to the normal class. This method is the desired behavior for most applications since it results in the best possible real-time performance. Win32 threads that contain GUI components may result in serious performance since real-time threads have a higher priority than Win32 threads in the real-time priority class. To have GUI-based Win32 threads scheduled on an equal basis as other threads, call **SetPriorityClass**.

Table 1, RTSS to Win32 NT Thread Priority Mapping, shows how the RTSS symbolic priority names translate to requests for a particular Windows NT and Windows 2000 priority when calling **RtSetThreadPriority** in a Win32 program.

For instance, **RtSetThreadPriority**(hThread, RT_PRIORITY_MIN+1) results in a call to **SetThreadPriority**(hThread,thread_priority_lowest) by the Win32 version of the RTX interfaces.

Table 1. RTSS to Win32 NT and Windows 2000 Thread Priority Mapping

RTSS Symbolic Priority Name	RTSS Value	Windows NT and Windows 2000 Symbolic Priority Name for Real-Time Priority Class	Win32 Value
RT_PRIORITY_MIN	0	THREAD_PRIORITY_IDLE	16
RT_PRIORITY_MIN + 1	1	THREAD_PRIORITY_LOWEST	22
RT_PRIORITY_MIN + 2	2	THREAD_PRIORITY_BELOW_NORMAL	23
RT_PRIORITY_MIN + 3	3	THREAD_PRIORITY_NORMAL	24
RT_PRIORITY_MIN + 4	4	THREAD_PRIORITY_ABOVE_NORMAL	25
RT_PRIORITY_MIN + 5...+ 126	5...126	THREAD_PRIORITY_HIGHEST	26
RT_PRIORITY_MAX	127	THREAD_PRIORITY_TIME_CRITICAL	31

Any value from RT_PRIORITY_MIN+5 to RT_PRIORITY_MIN+126 will put the thread at THREAD_PRIORITY_HIGHEST and RT_PRIORITY_MAX will result in the THREAD_PRIORITY_TIME_CRITICAL priority. These mappings are fixed and are designed to preserve relative ordering among thread priorities.

Win32 NT callers of **RtGetThreadPriority**( ) will have returned the real-time priority that was specified in the call to **RtSetThreadPriority**( ). There are some restrictions. The most likely source of confusion is when calls to **RtSetThreadPriority** and **SetThreadPriority** are mixed. The library may not always understand the RTSS priority when a duplicated thread handle is used. In these cases, the caller should expect that RT_PRIORITY_MIN+5 will be returned instead of RT_PRIORITY_MIN+6 through RT_PRIORITY_MIN+126. Threads that set and get their own RTSS priorities, i.e., specify the thread with **GetCurrentThread**( ), will always get the RTSS priority that was set.

Win32 programs should use the Rt versions of the priority calls if the Win32 thread wants to claim other than the lowest RTSS scheduling priority when waiting on RTSS synchronization objects. For instance, a Win32 thread with an RTSS priority of RT_PRIORITY_MAX will own a mutex before an RTSS thread waiting for the same mutex with a priority less than RT_PRIORITY_MAX.

Table 2, Win32 NT to RTSS Thread Priority Mapping, shows what callers of the Win32 set and get thread priority calls should expect in the RTSS environment. This table describes the inverse of the mapping shown in Table 1.

Table 2. Win32 NT and WIndows 2000 to RTSS Thread Priority Mapping

Windows NT and Windows 2000 Symbolic Priority Name for Real-Time Priority Class	Value	RTSS Symbolic Priority Name	Value
THREAD_PRIORITY_IDLE	16	RT_PRIORITY_MIN	0
THREAD_PRIORITY_LOWEST	22	RT_PRIORITY_MIN + 1	1
THREAD_PRIORITY_BELOW_NORMAL	23	RT_PRIORITY_MIN + 2	2
THREAD_PRIORITY_NORMAL	24	RT_PRIORITY_MIN + 3	3
THREAD_PRIORITY_ABOVE_NORMAL	25	RT_PRIORITY_MIN + 4	4
THREAD_PRIORITY_HIGHEST	26	RT_PRIORITY_MIN + 5	5
THREAD_PRIORITY_TIME_CRITICAL	31	RT_PRIORITY_MAX	127

There are no additional priorities between THREAD_PRIORITY_IDLE and THREAD_PRIORITY_HIGHEST. If the programmer needs finer grain priorities, then the RTSS priority spectrum should be used instead. The exception to this is when the value of THREAD_PRIORITY_TIME_CRITICAL is used. Just as in Win32, this value specifies a thread priority that is higher than all other priorities.

See Also

GetThreadPriority
RtSetThreadPriority
SetThreadPriority

RtGetThreadTimeQuantum

RtGetThreadTimeQuantum gets the current time quantum, in milliseconds, for the specified thread.

DWORD
RtGetThreadTimeQuantum(
HANDLE *hThread*
);

Paramenters

hThread

The handle for the specified thread, in milliseconds.

Return Values

If the function succeeds, the return value is the thread's time quantum in milliseconds. Otherwise, the return value is 0. To get extended error information, call **GetLastError**.

Comments

Win32 Environment: The only valid time quantum value is 0. This time quantum simulates Windows scheduling policy.

RTSS Environment: The time quantum can be any value greater than or equal to zero. A value of zero means the RTSS thread will run to completion.

The default time quantum value can be changed in the RTSS Control Panel.

See Also

RtSetThreadTimeQuantum

RtGetTimer

RtGetTimer returns the remaining relative time until the next expiration of the specified timer.

BOOL
RtGetTimer(
HANDLE *hTimer*,
PLARGE_INTEGER *pTimeRemaining*
);

Parameters

hTimer

An RTX-specific handle to the timer.

pTimeRemaining

A pointer to a LARGE_INTEGER structure in which to store the remaining time until expiration.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtGetTimer returns the relative amount of time until the specified timer expires. The time is specified in 100ns units and is written into the user-provided LARGE_INTEGER structure.

See Also

RtCreateTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTime
RtGetClockTimerPeriod
RtSetClockTime
RtSetTimer
RtSetTimerRelative

RtIsInRtss

RtIsInRtss returns TRUE if the calling process is running in the RTSS. Otherwise, the function returns FALSE.

BOOL
RtIsInRtss(VOID);

Parameters

This function has no parameters.

Return Values

The function returns TRUE if the calling process is running in the RTSS environment. Otherwise, this function returns FALSE.

Comments

A program should call this program if it must determine its run-time environment. The call returns FALSE when called from Win32.

RtLockKernel

RtLockKernel locks certain sections of Windows NT and Windows 2000 kernel's virtual address space into physical memory.

BOOL
RtLockKernel(
ULONG *Section*
);

Parameters

Section

A number specifying which section of the kernel to lock. Currently, RT_KLOCK_ALL is supported. This will lock down all pageable kernel sections.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtLockKernel locks down all specified eligible sections of the kernel's memory, so that it is not paged out, and so that it does not incur page faults during execution. Note that this should be done with caution, as the system performance may be greatly affected.

Since RTSS and Windows NT and Windows 2000 kernel portions of memory used by RTSS are always locked already, this function has no impact on RTSS determinism. This call is a NO-OP (no operation) when issued from RTSS applications.

See Also

RtAllocateLockedMemory
RtCommitLockHeap
RtCommitLockProcessHeap
RtCommitLockStack
RtFreeLockedMemory
RtLockProcess
RtUnlockKernel
RtUnlockProcess

RtLockProcess

RtLockProcess locks certain sections of a process' virtual address space into physical memory.

BOOL
RtLockProcess(
ULONG *Section*
);

Parameters

Section

A number specifying which sections to lock. Currently, RT_PLOCK_ALL is supported. This locks down all pageable process sections.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtLockProcess locks down all specified eligible committed memory sections of the process, so that it is not paged out, and so that it does not incur page faults during execution. Note that this should be done with caution, as the system performance may be greatly affected.

Since all RTSS processes are always locked, this function has no effect in the RTSS environment.

See Also

RtAllocateLockedMemory
RtCommitLockHeap
RtCommitLockProcessHeap
RtCommitLockStack
RtFreeLockedMemory
RtLockKernel
RtUnlockKernel
RtUnlockProcess

RtMapMemory

RtMapMemory maps a range of physical memory addresses into the user's virtual address space.

PVOID
RtMapMemory(
LARGE_INTEGER *physAddr*,
ULONG *Length*,
BOOLEAN *CacheEnable*
);

Parameters

physAddr

A LARGE_INTEGER specifying the base of the physical address range to map.

Length

An unsigned 32-bit value representing the length, in bytes, of the address range to map.

CacheEnable

A Boolean to indicate whether or not Windows NT and Windows 2000 should use the cache with this memory map.

Return Values

If successful, a virtual address in the calling process' space is returned. Because there are no access checks on these addresses or the ranges requested, care should be taken not to corrupt memory on the machine.

If the function fails, a NULL virtual address is returned.

Comments

RtMapMemory creates a map between a range of user virtual addresses and a range of physical memory addresses, giving the user direct access to physical memory locations on the system. Typically, this is used to access peripheral registers or buffers mapped into the physical address space of the machine. The largest address must be a legal value on the machine. For 32-bit machines, the largest address that can be represented is 0XFFFFFFFF.

See Also

RtAllocateLockedMemory
RtCommitLockHeap
RtCommitLockProcessHeap
RtCommitLockStack
RtFreeLockedMemory
RtLockKernel
RtLockProcess
RtUnlockKernel
RtUnlockProcess
RtUnmapMemory

RtOpenEvent

RtOpenEvent returns a handle of an existing named event object.

HANDLE
RtOpenEvent(
DWORD *DesiredAccess*,
BOOL *bInheritHandle*,
LPCTSTR *lpName*
);

Parameters

DesiredAccess (ignored)

bInheritHandle (ignored)

lpName

A pointer to a null-terminated string that names the event to be opened. Name comparisons are case-sensitive.

Return Values

If the function succeeds, the return value is a handle of the event object.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

RtOpenEvent enables multiple processes to open handles of the same event object. The function succeeds only if some process has already created the event by using **RtCreateEvent**. The calling process can use the returned handle in any function that requires a handle of an event object, such as a wait function.

Use **RtCloseHandle** to close the handle. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

See Also

RtCloseHandle
RtCreateEvent
RtPulseEvent
RtResetEvent
RtSetEvent

RtOpenMutex

RtOpenMutex returns a handle to the named RTSS mutex.

HANDLE
RtOpenMutex(
DWORD *DesiredAccess*,
BOOL *bInheritHandle*,
LPCTSTR *lpName*
);

Parameters

DesiredAccess (ignored)

bInheritHandle (ignored)

lpName

A pointer to a null-terminated string that names the mutex to be opened. Name comparisons are case-sensitive.

Return Values

If the function succeeds, the return value is a handle of the mutex object.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

RtOpenMutex enables multiple processes to open handles of the same mutex object. The function succeeds only if some process has already created the mutex with **RtCreateMutex**. The calling process can use the returned handle in any function that requires a handle of a mutex object, such as a wait function.

Use **RtCloseHandle** to close the handle. The system closes the handle automatically when the process terminates. The mutex object is destroyed when its last handle has been closed.

See Also

RtCloseHandle
RtCreateMutex
RtReleaseMutex

RtOpenProcess

RtOpenProcess returns a handle to an existing process object.

```
HANDLE
RtOpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
    );
```

Parameters

dwDesiredAccess (ignored)

Specifies the access to the process object.

bInheritHandle (ignored)

Specifies whether the returned handle can be inherited by a new process created by the current process. If TRUE, the handle is inheritable.

dwProcessId

The process identifier of the process to open.

Return Values

If the function succeeds, the return value is an open handle to the specified process.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

The handle returned by **RtOpenProcess** can be used in any function that requires a handle to a process, such as the wait functions, provided the appropriate access rights were requested.

When you are finished with the handle, be sure to close it using **CloseHandle**.

See Also

CloseHandle
CreateProcess
RtGetExitCodeProcess
RtTerminateProcess

RtOpenSemaphore

RtOpenSemaphore returns a handle of an existing named semaphore object.

HANDLE
RtOpenSemaphore(
DWORD *DesiredAccess*,
BOOL *bInheritHandle*,
LPCTSTR *lpName*
);

Parameters

DesiredAccess (ignored)

bInheritHandle

This must be FALSE.

lpName

A pointer to a null-terminated string that names the semaphore to be opened. Name comparisons are case-sensitive.

Return Values

If the function succeeds, the return value is a handle of the semaphore object.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

RtOpenSemaphore enables multiple processes to open handles of the same semaphore object. The function succeeds only if some process has already created the semaphore by using **RtCreateSemaphore**. The calling process can use the returned handle in any function that requires a handle of a semaphore object, such as a wait function, subject to the limitations of the access specified in *DesiredAccess*.

Use **RtCloseHandle** to close the handle. The system closes the handle automatically when the process terminates. The semaphore object is destroyed when its last handle has been closed.

See Also

RtCloseHandle
RtReleaseSemaphore

RtOpenSharedMemory

RtOpenSharedMemory opens a named physical-mapping object.

HANDLE
RtOpenSharedMemory(
DWORD *DesiredAccess*,
BOOL *bInheritHandle*,
LPCTSTR *lpName*,
VOID ** *location*
);

Parameters

DesiredAccess

The access mode. The RTSS environment always grants read and write access. This parameter can be one of the following values:

Value	Meaning
SHM_MAP_WRITE	Read-write access. The target shared memory object must have been created with PAGE_READWRITE protection. A read-write view of the shared memory is mapped.
SHM_MAP_READ	Read-only access. The target shared memory object must have been created with PAGE_READWRITE or PAGE_READ protection. A read-only view of the shared memory is mapped.

BInheritHandle (ignored)

lpName

A pointer to a string that names the shared memory object to be opened. If there is an open handle to a shared memory object by this name, the open operation succeeds.

location

A pointer to a location where the virtual address of the mapping will be stored.

Return Values

If the function succeeds, the return value is an open handle to the specified shared memory object.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

The handle that **RtOpenSharedMemory** returns can be used with **RtCloseHandle** to decrement the reference count to the shared memory object. When the reference count is zero, the object is removed from the system.

In the same process, different calls to **RtOpenSharedMemory** may return different locations because they are mapped into different virtual addresses.

See Also

RtCreateSharedMemory
RtCloseHandle

RtPrintf

RtPrintf prints formatted output to the standard output stream or console window.

INT
RtPrintf(
LPCSTR *lpFormat* [,argument, . . .]
);

Parameters

lpFormat

The format control (with optional arguments).

Return Values

RtPrintf returns the number of characters printed. If an error occurs, it returns a negative value.

Comments

RtPrintf is similar to **printf**, but **RtPrintf** does not require the C run-time library and can work with any combination of run-time libraries. This function does not support floating point conversions in the RTSS environment.

RtPrintf formats and prints a series of characters and values to the standard output stream, stdout. If arguments follow the format string, the format string must contain specifications that determine the output format for the arguments.

The format argument consists of ordinary characters, escape sequences, and (if arguments follow format) format specifications. The ordinary characters and escape sequences are copied to stdout in order of their appearance. The required header is <rtapi.h>.

Format specifications always begin with a percent sign (%) and are read left to right. When **RtPrintf** encounters the first format specification (if any), it converts the value of the first argument after format and outputs it accordingly. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

Example

```
RtPrintf("Line one\n\t\tLine two\n");
```

produces the output:

```
Line one
  Line two
```

Format Specification Fields (RtPrintf and RtWprintf)

A format specification, which consists of optional and required fields, has the following form:

%[*flags*] [*width*] .*precision*] [{h|l|L}]*type*

Each field of the format specification is a single character or number signifying a particular format option. The simplest format specification contains only the percent sign and a type character (for example, %s). If a percent sign is followed by a character that has no meaning as a format field, the character is copied to stdout. For example, to print a percent-sign character, use %%.

Required format field: The type character, which appears after the optional format fields, is the only required format field. It determines whether the associated argument is interpreted as a character, a string, or a number, as shown in the table that follows.

In the RTSS environment, the following limitations apply:

- The floating point format is not supported (e, E, and f).
- The maximum output size is 256 characters.
- For **RtPrintf** only, there is a limit of ten parameters.

Note: The types C and S, and the behavior of c and s with **RtPrintf** and **RtWprintf** are consistent with Microsoft extensions for **printf** and are not ANSI compatible.

Type Character	Argument Type	Output
c	int	For **RtPrintf**, specifies a single-byte character. For **RtWprintf**, specifies a wide character.
C	int	For **RtPrintf**, specifies a wide character. For **RtWprintf**, specifies a single-byte character.
d	int	Signed decimal integer.
i	int	Signed decimal integer.
u	int	Unsigned decimal integer.
x	int	Unsigned hexadecimal integer, using "abcdef."
X	int	Unsigned hexadecimal integer, using "ABCDEF."
e	double	Signed value in the form: [-]d.dddd e [sign]ddd *where*, d is a single decimal digit, ddd is one or more decimal digits, ddd is exactly three decimal digits, and the sign is + or -.
E	double	Same as the "e" specifier. See above.
f	double	Signed value in the form: [-]dddd.dddd *where*, dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
s	string	For **RtPrintf**, specifies a single-byte character string. For **RtWprintf**, specifies a wide character string.

Type Character	Argument Type	Output
		Characters are printed up to the first null character or until the precision value is reached.
S	string	For **RtPrintf**, specifies a wide character string. For **RtWprintf**, specifies a single-byte character string. Characters are printed up to the first null character or until the precision value is reached.

Optional format fields: The optional fields, which appear before the type character, control other aspects of the formatting, as shown in the list that follows.

flags
: Optional character(s) that control justification of output and print of signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag can appear in a format specification.*width*
: Optional number that specifies the minimum number of characters.

precision
: Optional number that specifies the maximum number of characters printed for all or part of the output field or the minimum number of digits printed for integer values.

H|l| L
: Optional prefixes to *type* that specify the size of the argument.

See Also

RtAtoi
RtWPrintf
RtWtoi

RtPulseEvent

RtPulseEvent provides a single operation that sets (to signaled) the state of the specified event object and then resets it (to non-signaled) after releasing the appropriate number of waiting threads.

BOOL
RtPulseEvent(
HANDLE *hEvent*
);

Parameters

hEvent

Identifies the event object. The **RtCreateEvent** or **RtOpenEvent** function returns this handle.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

For a manual-reset event object, all waiting threads that can be released are released. The function then resets the event object's state to non-signaled and returns.

For an auto-reset event object, the function resets the state to non-signaled and returns after releasing a single waiting thread, even if multiple threads are waiting.

If no threads are waiting, or if no thread can be released immediately, **RtPulseEvent** simply sets the event object's state to non-signaled and returns.

See Also

RtCreateEvent
RtOpenEvent
RtResetEvent
RtSetEvent

RtReadPortBufferUchar
RtReadPortBufferUshort
RtReadPortBufferUlong

RtReadPortBuffer* calls copy a series of one-, two-, or four-byte quanta from an I/O port to a buffer.

VOID
RtReadPortBufferUchar(PUCHAR *PortAddress*, **PUCHAR** *pBuffer*, **ULONG** *nNumberOfBytes*);
VOID
RtReadPortBufferUshort(PUSHORT *PortAddress*, **PUSHORT** *pBuffer*, **ULONG** *nNumberOfBytes*);
VOID
RtReadPortBufferUlong(PULONG *PortAddress*, **PULONG** *pBuffer*, **ULONG** *nNumberOfBytes*);

Parameters

PortAddress

A Port I/O address cast as a pointer to the type of data being read.

pBuffer

A pointer to a buffer.

nNumberOfBytes

The size of the buffer to be read.

Comments

RtReadPortBufferUchar, **RtReadPortBufferUshort**, and **RtReadPortBufferUlong** read a buffer of one-, two-, or four-byte quanta directly from an I/O port to a buffer.

See Also

RtDisablePortIo
RtEnablePortIo
RtReadPort* (Uchar, Ushort, Ulong)
RtWritePortBuffer* (Uchar, Ushort, Ulong)
RtWritePort* (Uchar, Ushort, Ulong)

RtReadPortUchar
RtReadPortUshort
RtReadPortUlong

RtReadPort* read a one-, two-, or four-byte quantum directly from an I/O port.

UCHAR
RtReadPortUchar(PUCHAR *PortAddress*);
USHORT
RtReadPortUshort(PUSHORT *PortAddress*);
ULONG
RtReadPortUlong(PULONG *PortAddress*);

Parameters

PortAddress

A Port I/O address cast as a pointer to the type of data to be read.

Comments

RtReadPortUchar, **RtReadPortUshort**, and **RtReadPortUlong** read a one-, two-, or four-byte quantum directly from an I/O port and return the value.

See Also

RtDisablePortIo
RtEnablePortIo
RtReadPortBuffer* (Uchar, Ushort, Ulong)
RtWritePortBuffer* (Uchar, Ushort, Ulong)
RtWritePort* (Uchar, Ushort, Ulong)

RtReleaseInterruptVector

RtReleaseInterruptVector releases a previously attached interrupt. This breaks the association between a user's interrupt handling routine and the hardware interrupt.

BOOL
RtReleaseInterruptVector(
HANDLE *hInterrupt*
);

Parameters

hInterrupt

An RTX-specific handle as returned by a preceding call to **RtAttachInterruptVector**.

Return Values

The function returns TRUE if successful, or it returns FALSE if the argument was invalid or the operation on the handle did not succeed.

Comments

RtReleaseInterruptVector breaks the association between a device interrupt and the user's handling routine. The user should take care to disable interrupt generation on the hardware device before making a call to this routine. Typically, this is done by writing to the command register of the device.

See Also

RtAttachInterruptVector
RtAttachInterruptVectorEx

RtReleaseMutex

RtReleaseMutex relinquishes ownership of an RTSS mutex.

BOOL
RtReleaseMutex(
HANDLE *hMutex*
);

Parameters

hMutex

The handle which identifies the mutex object as retuned by a preceding call to **RtCreateMutex** or **RtOpenMutex**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

RtReleaseMutex fails if the calling thread does not own the mutex object.

A thread gets ownership of a mutex by specifying a handle of the mutex in wait functions. The thread that creates a mutex object can also get immediate ownership without using one of the wait functions. When the owning thread no longer needs to own the mutex object, it calls **RtReleaseMutex**.

While a thread has ownership of a mutex, it can specify the same mutex in additional wait-function calls without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call **RtReleaseMutex** once for each time that the mutex satisfied a wait.

See Also

RtCreateMutex
RtOpenMutex
RtWaitForSingleObject

RtReleaseSemaphore

RtReleaseSemaphore increases the count of the specified semaphore object by a specified amount.

BOOL
RtReleaseSemaphore(
HANDLE *hSemaphore*,
LONG *lReleaseCount*,
LPLONG *lpPreviousCount*
);

Parameters

hSemaphore

The semaphore object. **RtCreateSemaphore** or **RtOpenSemaphore** returns this handle.

lReleaseCount

The amount by which the semaphore object's current count is to be increased. The value must be greater than zero. If the specified amount causes the semaphore's count to exceed the maximum count that was specified when the semaphore was created, the count is not changed and the function returns FALSE.

lpPreviousCount

A pointer to a 32-bit variable receives the previous count for the semaphore. This parameter can be NULL if the previous count is not required.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

The state of a semaphore object is signaled when its count is greater than zero and non-signaled when its count is equal to zero. The process that calls **RtCreateSemaphore** specifies the semaphore's initial count. Each time a waiting thread is released because of the semaphore's signaled state, the count of the semaphore is decreased by one.

Typically, an application uses a semaphore to limit the number of threads using a resource. Before a thread uses the resource, it specifies the semaphore handle in a call to one of the wait functions. When the wait function returns, it decreases the semaphore's count by one. When the thread has finished using the resource, it calls **RtReleaseSemaphore** to increase the semaphore's count by one.

Another use of **RtReleaseSemaphore** is during an application's initialization. The application can create a semaphore with an initial count of zero. This sets the semaphore's state to non-

signaled and blocks all threads from accessing the protected resource. When the application finishes its initialization, it uses **RtReleaseSemaphore** to increase the count to its maximum value to permit normal access to the protected resource.

RtReleaseSemaphore with high release counts (e.g., greater than 10), and **RtSetEvent** with **ManualReset** TRUE and a high number of waiting threads (e.g., greater than 10) will experience slightly longer latencies, which scale with the number of threads made run-able in the call. For the best determinism, developers should avoid designs that make a large number of threads run-able at one time.

Note: Actual time depends on the number of try-except frames and the amount of processing in except and finally routines, as specified by the application developer; RTX itself does not introduce any long latencies.

See Also

RtCreateSemaphore
RtOpenSemaphore

RtReleaseShutdownHandler

RtReleaseShutdownHandler destroys the shutdown handler object created by RtAttachShutdownHandler.

BOOL
RtReleaseShutdownHandler(
HANDLE *hShutdown*
);

Parameters

hShutdown

A handle returned by RtAttachShutdownHandler.

Return Values

The function returns TRUE when *hShutdown* specifies a valid shutdown handler object and the object has been successfully destroyed. Otherwise, the function returns FALSE and the caller may call **GetLastError** for more details.

Comments

The shutdown handler should not call **ExitThread** or **ExitProcess**, nor should it attempt to create or close any objects. The process will exit after the shutdown handler returns.

See Also

ExitThread
RtAttachShutdownHandler

RtResetEvent

RtResetEvent sets the state of the specified event object to non-signaled.

BOOL
RtResetEvent(
HANDLE *hEvent*
);

Parameters

hEvent

Identifies the event object. The **RtCreateEvent** or **RtOpenEvent** function returns this handle.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

The state of an event object remains non-signaled until it is explicitly set to signaled by the **RtSetEvent** or **RtPulseEvent** function. The non-signaled state blocks the execution of any threads that have specified the event object in a call to a wait function.

The **RtResetEvent** function is used primarily for manual-reset event objects, which must be set explicitly to the non-signaled state. Auto-reset event objects automatically change from signaled to non-signaled after a single waiting thread is released.

See Also

RtCreateEvent
RtOpenEvent
RtPulseEvent
RtSetEvent

RtSetBusDataByOffset

RtSetBusDataByOffset sets bus-configuration data for a device on a dynamically configurable I/O bus with a published, standard interface.

ULONG
RtSetBusDataByOffset(
BUS_DATA_TYPE *BusDataType*,
ULONG *BusNumber*,
ULONG *SlotNumber*,
PVOID *pBuffer*,
ULONG *Offset*,
ULONG *Length*
);

Parameters

BusDataType

The type of bus data to be set. Currently, its value can be **EisaConfiguration**, **Pos**, or **PCIConfiguration**.. The upper boundary on the bus types supported is always *MaximumBusDataType*.

BusNumber

The zero-based and system-assigned number of the bus in systems with more than one bus of the same *BusDataType*.

SlotNumber

The logical slot number. When PCIConfiguration is specified, this is a PCI_SLOT_NUMBER-TYPE value.

pBuffer

A pointer to a caller-supplied buffer with configuration information specific to *BusDataType*.
When PCIConfiguration is specified, the buffer contains some or all of the PCI_COMMON_CONFIG information for the given SlotNumber. The specified Offset and Length determine how much information is supplied. Certain members of PCI_COMMON_CONFIG have read-only values, and the caller is responsible for preserving the system-supplied values of read-only members.

Offset

The byte offset in the PCI_COMMON_CONFIG structure at which the caller-supplied configuration values begin. Callers can use the system-defined constant PCI_COMMON_HDR_LENGTH to specify the device-specific area of PCI_COMMON_CONFIG.

Length

The number of bytes in the buffer.

Return Values

RtSetBusDataByOffset returns the number of bytes of data successfully set for the given *SlotNumber*. If the given *BusDataType* is not valid for the current platform or if the supplied information is invalid, this routine returns zero.

Comments

When accessing the device-specific area of the PCI configuration space, **RtSetBusDataByOffset** guarantees the following:

- This routine never reads or writes data outside the range specified by the input Offset and Length.
- Even if the input Length is exactly a byte or a (two-byte) word, this routine never accesses any data outside the requested range.

The PCI structures and values above is defined in rtapi.h. For more information consult the *Device Driver Kit (DDK) for Windows NT and Windows 2000*.

See Also

HalSetBusDataByOffset (in the DDK for Windows NT and Windows 2000)

RtGetBusDataByOffset

RtTranslateBusAddress

RtSetClockTime

RtSetClockTime sets the current value of the specified clock.

```
BOOL
RtSetClockTime(
    CLOCK Clock,
    PLARGE_INTEGER pTime
    );
```

Parameters

Clock

A clock identifier.

pTime

A pointer to a LARGE_INTEGER structure specifying the new value for *Clock*.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtSetClockTime sets the value of the specified clock. The clock is specified in 100ns units. See the table in the **RtCreateTimer** section for a list of legal clock values.

See Also

RtCancelTimer
RtCreateTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTime
RtGetClockTimerPeriod
RtSetTimer
RtSetTimerRelative

RtSetEvent

RtSetEvent sets the state of the specified event object to signaled.

```
BOOL
RtSetEvent(
    HANDLE hEvent
    );
```

Parameters

hEvent

Identifies the event object. The RtCreateEvent or RtOpenEvent function returns this handle.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

The state of a manual-reset event object remains signaled until it is set explicitly to the non-signaled state by the **RtResetEvent** function. Any number of waiting threads, or threads that subsequently begin wait operations for the specified event object by calling the wait functions, can be released while the object's state is signaled.

The state of an auto-reset event object, the function resets the state to non-signaled and returns after releasing remains signaled until a single waiting thread is released, at which time the system automatically sets the state to non-signaled. If no threads are waiting, the event object's state remains signaled.

See Also

RtCreateEvent
RtOpenEvent
RtPulseEvent
RtResetEvent

RtSetThreadPriority

RtSetThreadPriority sets the priority value for the specified thread.

```
BOOL
RtSetThreadPriority(
    HANDLE hThread,
    int nPriority
    );
```

Parameters

hThread

The thread whose priority value is to be set.

nPriority

RTSS Environment: A priority level from 0 to 127, where 127 identifies the highest priority thread.

Win32 Environment: Win32 has only seven program-settable thread priorities in the real-time priority class. **RtSetThreadPriority** maps the 128 thread priorities into these seven priorities. The RTX Win32 library maintains the real-time thread priority and returns this value when **RtGetThreadPriority**() is called.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

See the Comments section in **RtGetThreadPriority** for details on the relationship between Win32 and RTSS thread priorities.

See Also

GetThreadPriority
RtGetThreadPriority
SetThreadPriority

RtSetThreadTimeQuantum

RtSetThreadTimeQuantum sets the time quantum for the specified thread.

```
BOOL
RtSetThreadTimeQuantum(
    HANDLE hThread,
    DWORD dwQuantumInMS
    );
```

Parameters

hThread

The handle of the thread whose time quantum is to be set.

dwQuantumInMS

A new time quantum value in milliseconds. The amount of time the thread will run before it yields to another RTSS thread with the same priority.

Return Values

If the function succeeds, the return value is FALSE.

If the function fails, the return value is TRUE. To get extended information, call **GetLastError**.

Comments

Win32 Environment: The only valid time quantum value is 0. This time quantum simulates Windows scheduling policy.

RTSS Environment: The time quantum can be any value greater than or equal to zero. A value of zero means the RTSS thread will run to completion.

The default time quantum value can be changed in the RTSS Control Panel.

See Also

RtGetThreadTimeQuantum

RtSetTimer

RtSetTimer sets the expiration time and repeat interval on the specified timer.

```
BOOL
RtSetTimer(
    HANDLE hTimer,
    PLARGE_INTEGER pExpiration,
    PLARGE_INTEGER pInterval
    );
```

Parameters

hTimer

An RTX-specific handle to the timer.

pExpiration

A pointer to a LARGE_INTEGER structure indicating the absolute time for the initial expiration of the timer. The clock is specified in 100ns units. If the value of the expiration time is less than zero, the call is interpreted as a request to set the timer relative to current time on the associated clock. The result is identical to calling **RtSetTimerRelative**, with the absolute value of the specified expiration time.

pInterval

A pointer to a LARGE_INTEGER structure indicating the amount of time between the first expiration and each successive expiration. The clock is specified in 100ns units.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtSetTimer sets the expiration time and repeat interval for the specified timer. If the repeat interval is non-zero, then after the first expiration, the timer will repeatedly expire at the specified interval. If the repeat interval pointer is NULL, then the timer will expire only once, i.e., it is a "one-shot" timer. Likewise, a non-NULL interval pointer may be passed in, with its value set to zero, for a one-shot timer.

Upon each expiration of the timer, the handling thread is signaled to indicate the expiration, and the specified handling routine is run. The timer signals expirations only on the RTX timer interrupt boundaries. The RTX timer interval will be rounded up to the RTX timer resolution. The highest RTX timer resolution is 100ms, which can be set in RTX Control Panel.

To reset the expiration of a timer that has been previously set, the user must ensure that the timer is not active. That is, it must be either a one-shot timer that has expired, or the user must first cancel the timer with **RtCancelTimer**.

See Also

RtCancelTimer
RtCreateTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTime
RtGetClockTimerPeriod
RtGetTimer
RtSetClockTime
RtSetTimerRelative

RtSetTimerRelative

RtSetTimerRelative sets the expiration time and repeat interval on the specified timer.

```
BOOL
RtSetTimerRelative(
    HANDLE hTimer,
    PLARGE_INTEGER pExpiration,
    PLARGE_INTEGER pInterval
    );
```

Parameters

hTimer

An RTX-specific handle to the timer.

pExpiration

A pointer to a LARGE_INTEGER structure indicating the time for the initial expiration of the timer. Expiration is calculated relative to the current value of the clock associated with the timer at creation. The clock is specified in 100ns units.

pInterval

A pointer to a LARGE_INTEGER structure indicating the amount of time between the first expiration and each successive expiration. The clock is specified in 100ns units.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtSetTimerRelative sets the relative expiration time and repeat interval for the specified timer. If the repeat interval is non-zero, then after the first expiration, the timer will repeatedly expire at the specified interval. If the repeat interval pointer is NULL, then the timer will expire only once, i.e., it is a "one-shot" timer. Likewise, a non-NULL interval pointer may be passed in, with its value set to zero, for a one-shot timer.

Upon each expiration of the timer, the handling thread is signaled to indicate the expiration, and the specified handling routine is run. The timer signals expirations only on the RTX timer interrupt boundaries. The RTX timer interval will be rounded up to the RTX timer resolution. The highest RTX timer resolution is 100ms, which can be set in RTX Control Panel.

To reset the expiration of a timer that has been previously set, the user must ensure that the timer is not active. That is, it must be either a one-shot timer that has expired, or the user must first cancel the timer with **RtCancelTimer**.

See Also

RtCancelTimer
RtCreateTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTime
RtGetClockTimerPeriod
RtGetTimer
RtSetClockTime
RtSetTimer

RtSleepFt

RtSleepFt suspends the current thread for the specified time.

VOID
RtSleepFt(
PLARGE_INTEGER *pDuration*
);

Parameters

pDuration

A pointer to a LARGE_INTEGER structure indicating the amount of time to sleep, in 100ns units. pDuration must be less than or equal to one second, and must be greater than or equal to the minimum timer period of the system.

Return Values

This function returns no value.

Comments

RtSleepFt suspends the given thread from execution for the specified amount of time.

An expiration interval of 0 yields the process to other equal priority runnable threads (if any).

See Also

RtCreateTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTime
RtGetClockTimerPeriod
RtGetTimer
RtSetClockTime
RtSetTimer
RtSetTimerRelative
Sleep

RtTranslateBusAddress

RtTranslateBusAddress translates a bus-specific address into the corresponding system logical address.

BOOL
RtTranslateBusAddress(
INTERFACE_TYPE *InterfaceType*,
ULONG *BusNumber*,
LARGE_INTEGER *BusAddress*,
PULONG *pAddressSpace*,
PLARGE_INTEGER *pTranslatedAddress*
);

Parameters

InterfaceType

The bus interface type, which can be one of the following: **Internal**, **ISA**, **EISA**, **MicroChannel**, **TurboChannel**, or **PCIBus**. The upper bound on the types of buses supported is always *MaximumInterfaceType*.

BusNumber

The zero-based and system-assigned bus number for the device is used together with *InterfaceType* to identify the bus for systems with more than one bus of the same type.

BusAddress

The bus-relative address.

pAddressSpace

A pointer that specifies whether the address is a port number or a memory address: **pAddressSpace* 0x0 indicates memory, 0X1 indicates I/O space.

pTranslatedAddress

A pointer to the translated address.

Return Values

A return value of TRUE indicates the system logical address corresponding to the given *BusNumber* and *BusAddress* has been returned in *pTranslatedAddress*.

Comments

There are many ways to connect a peripheral bus into a system. The memory address space of the bus, referred to as the logical address space, can be directly merged with the physical address space of the host, or some mapping may be involved. Also, some machines can have more than one bus, or a bus can have more than one address space, as in having separate memory and I/O addresses.

RtTranslateBusAddress performs this translation. The parameters to this routine include a bus number to support platforms with more than one bus of the same *InterfaceType*, the bus address to be translated, and a *pAddressSpace* specifier typically used to differentiate between memory and I/O space, if these are separate. The user might obtain these parameters by calling **RtGetBusDataByOffset**.

See Also

HalTranslateBusAddress (in the DDK for Windows NT and Windows 2000)
RtGetBusDataByOffset
RtSetBusDataByOffset

RtTerminateProcess

RtTerminateProcess terminates the specified process and all of its threads.

```
BOOL
RtTerminateProcess(
    HANDLE hProcess,
    UINT uExitCode
    );
```

Parameters

hProcess

Handle to the process to terminate. The handle must have PROCESS_TERMINATE access.

uExitCode

The exit code for the process and for all threads terminated as a result of this call. Use **RtGetExitCodeProcess** to retrieve the process's exit value. Use **GetExitCodeThread** to retrieve a thread's exit value.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

RtTerminateProcess is used to unconditionally cause a process to exit. **Use it only in extreme circumstances.** The state of global data maintained by dynamic-link libraries (DLLs) may be compromised if **RtTerminateProcess** is used rather than **ExitProcess**.

RtTerminateProcess causes all threads within a process to terminate, and causes a process to exit, but DLLs attached to the process are not notified that the process is terminating.

Terminating a process causes the following:

1. All of the object handles opened by the process are closed.
2. All of the threads in the process terminate their execution.
3. The state of the process object becomes signaled, satisfying any threads that had been waiting for the process to terminate.
4. The states of all threads of the process become signaled, satisfying any threads that had been waiting for the threads to terminate.
5. The termination status of the process changes from STILL_ACTIVE to the exit value of the process.

Terminating a process does not cause child processes to be terminated.

Terminating a process does not necessarily remove the process object from the system. A process object is deleted when the last handle to the process is closed.

See Also

ExitProcess
RtGetExitCodeProcess
GetExitCodeThread
RtOpenProcess
RtCreateProcess

RtUnlockKernel

RtUnlockKernel unlocks sections of the kernel's virtual address space previously locked into physical memory.

BOOL
RtUnlockKernel(
    ULONG *Section*
    );

Parameters

Section

A number specifying which section of the kernel to lock. Currently, the value RT_KLOCK_ALL is supported.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtUnlockKernel unlocks sections of the kernel's memory previously locked by a call to **RtLockKernel**. The behavior of unlocking memory not previously locked by **RtLockKernel** is undefined and unpredictable.

Because processes are always locked, this function has no impact on RTSS determinism. This call is a NO-OP (no operation) when issued from RTSS applications.

See Also

RtAllocateLockedMemory
RtCommitLockHeap
RtCommitLockProcessHeap
RtCommitLockStack
RtFreeLockedMemory
RtLockKernel
RtLockProcess
RtUnlockProcess

RtUnlockProcess

RtUnlockProcess unlocks sections of the processes' virtual address space previously locked into physical memory. **RtUnlockProcess** has no effect within the RTSS environment.

```
BOOL
RtUnlockProcess(
    ULONG Section
    );
```

Parameters

Section

A number specifying which sections to unlock. Currently, the value RT_PLOCK_ALL is supported.

Return Values

The function returns TRUE if it completes successfully, or it returns FALSE if invalid parameters are specified.

Comments

RtUnlockProcess unlocks sections of the process' memory previously locked by a call to RtLockProcess. The behavior of unlocking memory not previously locked by RtLockProcess is undefined and unpredictable. This function has no impact on RTSS determinism. This call is a NO-OP (no operation) when issued from RTSS applications.

See Also

RtAllocateLockedMemory
RtCommitLockHeap
RtCommitLockProcessHeap
RtCommitLockStack
RtFreeLockedMemory
RtLockKernel
RtLockProcess
RtUnlockKernel

RtUnmapMemory

RtUnmapMemory releases a mapping made by a previous call to **RtMapMemory**.

```
BOOL
RtUnmapMemory(
    PVOID pVirtualAddress
    );
```

Parameters

pVirtualAddress

A pointer returned by a previous call to **RtMapMemory**.

Return Values

If the function is successful, it returns TRUE. Otherwise, it returns FALSE.

Comments

The virtual address passed in must be the same base address returned to the user by a previous call to **RtMapMemory**.

See Also

RtAllocateLockedMemory
RtCommitLockHeap
RtCommitLockProcessHeap
RtCommitLockStack
RtFreeLockedMemory
RtLockKernel
RtLockProcess
RtMapMemory
RtUnlockKernel
RtUnlockProcess

RtWaitForMultipleObjects

RtWaitForMultipleObjects returns when one of the following occurs:

- Any one of the specified objects is in the signaled state.
- The time-out interval elapses.
- This function only supports *WAIT FOR ANY* object.

DWORD
RtWaitForMultipleObjects(
DWORD *nCount*,
CONST HANDLE **lpHandles*,
BOOL *fWaitAll*,
DWORD *dwMilliseconds*
);

Parameters

nCount

Specifies the number of object handles in the array pointed to by *lpHandles*. The maximum number of object handles is MAX_WFMO, as defined in RTAPI.h.

lpHandles

Pointer to an array of object handles. For a list of the object types whose handles can be specified, see the following Remarks section. The array can contain handles to objects of different types. It may not contain the multiple copies of the same handle.

If one of these handles is closed while the wait is still pending, the function's behavior is undefined.

The handles must have SYNCHRONIZE access.

fWaitAll

Specifies the wait type. The function returns when the state of **any one** of the objects set to is signaled. The return value indicates the object whose state caused the function to return.

RTSS Environment: This parameter must be FALSE, indicating WAIT FOR ANY.

dwMilliseconds

Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the condition specified by the *fWaitAll* parameter are not met. If *dwMilliseconds* is zero, the function tests the states of the specified objects and returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return. This value can be one of the following.

Return Value	Meaning
WAIT_OBJECT_0 to (WAIT_OBJECT_0 + nCount - 1)	The return value minus WAIT_OBJECT_0 indicates the lpHandles array index of the object that satisfied the wait. If more than one object became signalled during the call, this is the array index of the signalled object with the smallest index value of all the signalled objects.
WAIT_ABANDONED_0 to (WAIT_ABANDONED_0 + nCount - 1)	The return value minus WAIT_ABANDONED_0 indicates the *lpHandles* array index of an abandoned mutex object that satisfied the wait.
WAIT_TIMEOUT	The time-out interval elapsed and the conditions specified by the *fWaitAll* parameter are not satisfied. If the function fails, the return value is WAIT_FAILED. To get extended error information, call **GetLastError**.

Remarks

RtWaitForMultipleObjects determines whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. For example, the count of a semaphore object is decreased by one. When *fWaitAll* is FALSE, and multiple objects are in the signaled state, the function chooses one of the objects to satisfy the wait; the states of the other objects are unaffected.

RtWaitForMultipleObjects can specify handles of any of the following object types in the *lpHandles* array:

- Event
- Mutex
- Process
- Semaphore
- Thread

See Also

CreateThread
RtOpenProcess
RtCreateEvent
RtCreateMutex
RtCreateProcess
RtCreateSemaphore
RtOpenEvent
RtOpenMutex
RtOpenSemaphore
RtWaitForSingleObject

RtWaitForSingleObject

RtWaitForSingleObject returns when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses.

DWORD
RtWaitForSingleObject(
HANDLE *Handle*,
DWORD *Milliseconds*
);

Parameters

hHandle

The object identifier. See the list of the object types whose handles can be specified in the Comments section.

Milliseconds

The time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is non-signaled. If *Milliseconds* is zero, the function tests the object's state and returns immediately. If *Milliseconds* is INFINITE, the function's time-out interval never elapses.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return.

If the function fails, the return value is WAIT_FAILED. To get extended error information, call **GetLastError**.

The return value on success is one of the following values:

Value	Meaning
WAIT_ABANDONED	The specified object is a mutex object that was not released by the thread that owned the mutex object before the owning thread terminated. Ownership of the mutex object is granted to the calling thread, and the mutex is set to non-signaled.
WAIT_OBJECT_0	The state of the specified object is signaled.
WAIT_TIMEOUT	The time-out interval elapsed, and the object's state is non-signaled.

Comments

RtWaitForSingleObject checks the current state of the specified object. If the object's state is non-signaled, the calling thread enters an efficient wait state. The thread consumes very little processor time while waiting for the object state to become signaled or the time-out interval to elapse.

Before returning, a wait function modifies the state of some types of synchronization objects.

Modification occurs only for the object or objects whose signaled state caused the function to return. For example, the count of a semaphore object is decreased by one.

RtWaitForSingleObject can wait for the following objects:

Semaphore
RtCreateSemaphore or **RtOpenSemaphore** returns the handle. A semaphore object maintains a count between zero and some maximum value. Its state is signaled when its count is greater than zero and non-signaled when its count is zero. If the current state is signaled, the wait function decreases the count by one.

Mutex
RtCreateMutex and **RtOpenMutex** return handles to the mutex object which becomes signaled when the mutex is unowned.

Event
The **RtCreateEvent** or **RtOpenEvent** function returns the handle. An event object's state is set explicitly to signaled by the **RtSetEvent** or **RtPulseEvent** function. A manual-reset event object's state must be reset explicitly to nonsignaled by the **RtResetEvent** function. For an auto-reset event object, the wait function resets the object's state to nonsignaled before returning.

See Also

RtCreateEvent
RtCreateMutex
RtCreateSemaphore
RtOpenEvent
RtOpenMutex
RtOpenSemaphore

RtWprintf

RtWprintf prints formatted output to the standard output stream or console window.

INT
RtWprintf(
LPCWSTR *lpFormat* [, argument, . . .]
);

Parameters

lpFormat

The format control with optional arguments.

Return Values

RtWprintf returns the number of wide characters printed. If an error occurs, it returns a negative value.

Comments

RtWprintf is similar to **wprintf**, but **RtWprintf** does not require the C run-time library and can work with any combination of run-time libraries. This function does not support floating point conversions in the RTSS environment.

RtWprintf formats and prints a series of characters and values to the standard output stream, stdout. If arguments follow the format string, the format string must contain specifications that determine the output format for the arguments.

The format argument consists of ordinary characters, escape sequences, and (if arguments follow format) format specifications. The ordinary characters and escape sequences are copied to stdout in order of their appearance. The required header is <rtapi.h>.

Format specifications always begin with a percent sign (%) and are read left to right. When **RtWprintf** encounters the first format specification (if any), it converts the value of the first argument after format and outputs it accordingly. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

Example

```
RtWprintf(L"Line one\n\t\tLine two\n");
```

produces the output:

```
Line one
  Line two
```

Format Specification Fields

See the format specification fields in the Comments section of **RtPrintf** for details.

See Also

RtAtoi
RtPrintf
RtWtoi

RtWritePortBufferUchar
RtWritePortBufferUshort
RtWritePortBufferUlong

The RtWritePortBuffer* calls copy a series of one-, two-, or four-byte quanta from a buffer to an I/O port.

VOID
RtWritePortBufferUchar(PUCHAR *PortAddress*, **PUCHAR** *pBuffer*, **ULONG** *nNumberOfBytes*);
VOID
RtWritePortBufferUshort(PUSHORT *PortAddress*, **PUSHORT** *pBuffer*, **ULONG** *nNumberOfBytes*);
VOID
RtWritePortBufferUlong(PULONG *PortAddress*, **PULONG** *pBuffer*, **ULONG** *nNumberOfBytes*);

Parameters

PortAddress

A Port I/O address cast as a pointer to the type of data being written.

pBuffer

A pointer to a buffer of one-, two-, or four-byte quanta.

nNumberOfBytes

The size of the buffer to be written.

Comments

RtWritePortBufferUchar, **RtWritePortBufferUshort**, and **RtWritePortBufferUlong** write a buffer of one-, two-, or four-byte quanta directly to an I/O port from the buffer.

See Also

RtDisablePortIo
RtEnablePortIo
RtReadPortBuffer* (Uchar, Ushort, Ulong)
RtReadPort* (Uchar, Ushort, Ulong)
RtWritePort* (Uchar, Ushort, Ulong)

RtWritePortUchar
RtWritePortUshort
RtWritePortUlong

The RtWritePort* calls write a one-, two-, or four-byte quantum directly to an I/O port.

VOID
RtWritePortUchar(PUCHAR *PortAddress*, **UCHAR** *pBuffer*);
VOID
RtWritePortUshort(PUSHORT *PortAddress*, **USHORT** *pBuffer*);
VOID
RtWritePortUlong(PULONG *PortAddress*, **ULONG** *pBuffer*);

Parameters

PortAddress

A Port I/O address cast as a pointer to the type of data being written.

pBuffer

The one-, two-, or four-byte quantum to be written to the port.

Comments

RtWritePortUchar, **RtWritePortUshort**, and **RtWritePortUlong** write a one-, two-, or four-byte quantum directly to an I/O port.

See Also

RtDisablePortIo
RtEnablePortIo
RtReadPortBuffer* (Uchar, Ushort, Ulong)
RtReadPort* (Uchar, Ushort, Ulong)
RtWritePortBuffer* (Uchar, Ushort, Ulong)

RtWtoi

RtWtoi converts a given string value to an integer.

```
INT
RtWtoi(
    LPCWSTR lpString
    );
```

Parameters

lpString

The source Unicode string.

Return Values

This function returns the integer value of the string.

Comments

RtWtoi is similar to **wtoi**, but **RtWtoi** does not require the C run-time library and can work with any combination of run-time libraries.

This function supports decimal digits only, and does not allow leading whitespace or signs.

See Also

RtAtoi
RtWPrintf

CHAPTER 3

Win32-Supported API

AbnormalTermination

AbnormalTermination indicates whether the try block of a try-finally statement terminated normally. The function can be called only from within the finally block of a try-finally statement.

BOOL
AbnormalTermination(VOID)

Parameters

This function has no parameters.

Return Values

If the try block of the try-finally statement terminated abnormally, the return value is TRUE.

If the try block of the try-finally statement terminated normally, the return value is FALSE.

Comments

The try block terminates normally only if execution leaves the block sequentially after executing the last statement in the block. Statements (such as **return**, **goto**, **continue**, or **break**) that cause execution to leave the try block result in abnormal termination of the block. This is the case even if such a statement is the last statement in the try block.

Abnormal termination of a try block causes the system to search backward through all stack frames to determine whether any termination handlers must be called. This can result in the execution of hundreds of instructions, so it is important to avoid abnormal termination of a try block due to a **return**, **goto**, **continue**, or **break** statement. Note that these statements do not generate an exception, even though the termination is abnormal.

CloseHandle

CloseHandle closes an open object handle.

```
BOOL
CloseHandle(
    HANDLE hObject
    );
```

Parameters

hObject

An open object handle.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

CloseHandle closes handles to thread and file objects. It invalidates the specified object handle, decrements the object's handle count, and performs object-retention checks. Once the last handle to an object is closed, the object is removed from the operating system.

Note: In the RTSS environment, **CloseHandle** can also be used to close any RTX object. Use the **RtCloseHandle** to close RTX objects. In the Win32 environment, **CloseHandle** can only close Win objects.

See Also

RtCloseHandle

CreateDirectory

CreateDirectory creates a new directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory.

BOOL
CreateDirectory(
LPCTSTR *lpPathName*,
LPSECURITY_ATTRIBUTES *lpSecurityAttributes*
);

Parameters

lpPathName

A pointer to a null-terminated string that specifies the path of the directory to be created.

There is a default string size limit for paths of RTX_MAX_PATH characters. This limit is related to how **CreateDirectory** parses paths.

lpSecurityAttributes

A pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If *lpSecurityAttributes* is NULL, the handle cannot be inherited.

The *lpSecurityDescriptor* member of the structure specifies a security descriptor for the new directory. If *lpSecurityAttributes* is NULL, the directory gets a default security descriptor. The target file system must support security on files and directories for this parameter to have an effect.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

Some file systems, such as NTFS, support compression for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression attribute of its parent directory.

See Also

CreateFile

CreateFile

CreateFile creates or opens two types of objects: files and directories (open only). It then returns a handle that can be used to access the object.

```
HANDLE
CreateFile(
    LPCTSTR lpFileName,
    DWORD DesiredAccess,
    DWORD ShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD CreationDisposition,
    DWORD FlagsAndAttributes,
    HANDLE hTemplateFile
    );
```

Parameters

lpFileName

A pointer to a null-terminated string that specifies the name of the object to create or open.

If **lpFileName* is a path, the string size limit is RTX_MAX_PATH characters. This limit is related to how **CreateFile** parses paths.

DesiredAccess

The type of access to the object. An application can obtain read access, write access, read-write access, or device query access. This parameter can be any combination of the following values.

0 (zero)

Specifies device query access to the object. An application can query device attributes without accessing the device.

GENERIC_READ

Specifies read access to the object. Data can be read from the file and the file pointer can be moved. Combine with GENERIC_WRITE for read-write access.

GENERIC_WRITE

Specifies write access to the object. Data can be written to the file and the file pointer can be moved. Combine with GENERIC_READ for read-write access.

ShareMode

Set of bit flags that specifies how the object can be shared. If ShareMode is 0, the object cannot be shared. Subsequent open operations on the object will fail, until the handle is closed.

To share the object, use a combination of one or more of the following values:

FILE_SHARE_DELETE
Subsequent open operations on the object will succeed only if delete access is requested.

FILE_SHARE_READ
Subsequent open operations on the object will succeed only if read access is requested.

FILE_SHARE_WRITE
Subsequent open operations on the object will succeed only if write access is requested.

lpSecurityAttributes (ignored by RTX)

A pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If *lpSecurityAttributes* is NULL, the handle cannot be inherited.

The *lpSecurityDescriptor* member of the structure specifies a security descriptor for the object. If *lpSecurityAttributes* is NULL, the object gets a default security descriptor. The target file system must support security on files and directories for this parameter to have an effect on files.

CreationDisposition

Specifies which action to take on files that exist, and which action to take when files do not exist. For more information about this parameter, see the Comments section. This parameter must be one of the following values:

CREATE_NEW
Creates a new file. **CreateFile** fails if the specified file already exists.

CREATE_ALWAYS
Creates a new file. If the file exists, **CreateFile** overwrites the file and clears the existing attributes.

OPEN_EXISTING
Opens the file. **CreateFile** fails if the file does not exist. See the Comments section for information on hen to use the OPEN_EXISTING flag if using **CreateFile** for devices, including the console.

OPEN_ALWAYS
Opens the file, if it exists. If the file does not exist, **CreateFile** creates the file as if *CreationDisposition* were CREATE_NEW.

TRUNCATE_EXISTING
Opens the file. Once opened, the file is truncated so that its size is zero bytes. The calling process must open the file with at least GENERIC_WRITE access. **CreateFile** fails if the file does not exist.

FlagsAndAttributes

The file attributes and flags for the file.

Valid Attributes

Any combination of the following attributes is acceptable for the *FlagsAndAttributes* parameter, except all other file attributes override FILE_ATTRIBUTE_NORMAL.

FILE_ATTRIBUTE_ARCHIVE
The file should be archived. Applications use this attribute to mark files for backup or removal.

FILE_ATTRIBUTE_HIDDEN
The file is hidden. It is not to be included in an ordinary directory listing.

FILE_ATTRIBUTE_NORMAL
The file has no other attributes set. This attribute is valid only if used alone.

FILE_ATTRIBUTE_OFFLINE
The data of the file is not immediately available. Indicates that the file data has been physically moved to offline storage.

FILE_ATTRIBUTE_READONLY
The file is read only. Applications can read the file but cannot write to it or delete it.

FILE_ATTRIBUTE_SYSTEM
The file is part of or is used exclusively by the operating system.

FILE_ATTRIBUTE_TEMPORARY
The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

Valid Flags

Any combination of the following flags is acceptable for the *FlagsAndAttributes* parameter.

FILE_FLAG_WRITE_THROUGH
Instructs the system to write through any intermediate cache and go directly to disk. Windows can still cache write operations, but cannot lazily flush them.

FILE_FLAG_NO_BUFFERING
Instructs the system to open the file with no intermediate buffering or caching. An application must meet certain requirements when working with files opened with FILE_FLAG_NO_BUFFERING:

- File access must begin at byte offsets within the file that are integer multiples of the volume's sector size.
- File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.
- Buffer addresses for read and write operations must be aligned on addresses in memory that are integer multiples of the volume's sector size.

FILE_FLAG_RANDOM_ACCESS
Indicates that the file is accessed randomly. The system can use this as a hint to optimize file caching.

FILE_FLAG_SEQUENTIAL_SCAN
Indicates that the file is to be accessed sequentially from beginning to end. The system can use this as a hint to optimize file caching. If an application moves the file pointer for random access, optimum caching may not occur; however, correct operation is still guaranteed. Specifying this flag can increase performance for applications that read large

files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip over small ranges of bytes.

FILE_FLAG_DELETE_ON_CLOSE
Indicates that the operating system is to delete the file immediately after all of its handles have been closed, not just the handle for which you specified FILE_FLAG_DELETE_ON_CLOSE. Subsequent open requests for the file will fail, unless FILE_SHARE_DELETE is used.

hTemplateFile (ignored)

Return Values

If **CreateFile** succeeds, the return value is an open handle to the specified file. If the specified file exists before the function call and CreationDisposition is CREATE_ALWAYS or OPEN_ALWAYS, a call to **GetLastError** returns ERROR_ALREADY_EXISTS (even though the function has succeeded). If the file does not exist before the call, **GetLastError** returns ERROR_SUCCESS. If **CreateFile** fails, the return value is INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

Comments

Use **CloseHandle** to close an object handle returned by **CreateFile**.

As noted above, specifying zero for *DesiredAccess* allows an application to query device attributes without actually accessing the device. This type of querying is useful, for example, if an application wants to determine the size of a floppy disk drive and the formats it supports without having a floppy in the drive.

Files

When creating a new file, **CreateFile** performs the following actions:

- Combines the file attributes and flags specified by *FlagsAndAttributes* with FILE_ATTRIBUTE_ARCHIVE.
- Sets the file length to zero.
- Copies the extended attributes supplied by the template file to the new file if the *hTemplateFile* parameter is specified.

When opening an existing file, **CreateFile** performs the following actions:

- Combines the file flags specified by *FlagsAndAttributes* with existing file attributes. **CreateFile** ignores the file attributes specified by *FlagsAndAttributes*.
- Sets the file length according to the value of *CreationDisposition*.
- Ignores the *hTemplateFile* parameter.
- Ignores the *lpSecurityDescriptor* member of the SECURITY_ATTRIBUTES structure if the *lpSecurityAttributes* parameter is not NULL. The other structure members are used. bInheritHandle is the only way to indicate whether the file handle can be inherited.

Some file systems, such as NTFS, support compression for individual files and directories. On volumes formatted for such a file system, a new file inherits the compression attribute of its directory.

Do not use **CreateFile** to set a file's compression state. Setting FILE_ATTRIBUTE_COMPRESSED in the *FlagsAndAttributes* parameter does nothing. Use **DeviceIoControl** and the FSCTL_SET_COMPRESSION operation to set a file's compression state.

Directories

An application cannot create a directory with **CreateFile**; it must call **CreateDirectory** to create a directory.

See Also

CreateDirectory
DeviceIoControl
ReadFile
WriteFile

CreateThread

CreateThread creates a thread to execute within the address space of the calling process.

```
HANDLE
CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    DWORD StackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
    );
```

Parameters

lpThreadAttributes (ignored)

StackSize

The size, in bytes, of the stack for the new thread. If 0 is specified, the stack size defaults to 8192 bytes under RTSS or to the same size as the calling thread's stack under Win32. The stack is allocated automatically in the memory space of the process, and it is freed when the thread terminates. In the Win32 environment, the stack size grows when necessary. In the RTSS environment, the stack **cannot** grow.

The number of bytes specified by *StackSize* must be available from non-paged memory in the kernel.

lpStartAddress

A pointer to the application-supplied function to be executed by the thread and represents the starting address of the thread. The function accepts a single 32-bit argument and returns a 32-bit exit value.

lpParameter

A single 32-bit parameter value passed to the thread.

CreationFlags

Additional flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state and will not run until **ResumeThread** is called. If this value is zero, the thread runs immediately after creation. At this time, no other values are supported.

lpThreadId

A pointer to a 32-bit variable that receives the thread identifier.

Return Values

If the function succeeds, the return value is a handle to the new thread.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

The new thread handle is created with full access to the new thread.

The thread execution begins at the function specified by *lpStartAddress*. If this function returns, the DWORD return value is used to terminate the thread in an implicit call to **ExitThread**.

The thread is created with a thread priority of 0. Use **GetThreadPriority** and **SetThreadPriority** to get and set the priority value of a thread.

The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to **CloseHandle**. (For threads, use **CloseHandle** rather than **RtCloseHandle**.)

ExitProcess, **ExitThread**, and **CreateThread**, as well as a process that is starting, are serialized between each other within a process. Only one of these events can happen in an address space at a time.

See Also

CloseHandle
ExitProcess
ExitThread
GetThreadPriority
ResumeThread
SetThreadPriority

DeleteCriticalSection

DeleteCriticalSection releases all resources used by an unowned critical-section object.

VOID
DeleteCriticalSection(
LPCRITICAL_SECTION *lpCriticalSection*
);

Parameter

lpCriticalSection

A pointer to the critical-section object.

Return Values

This function does not return a value.

Comments

Deleting a critical-section object releases all system resources used by the object. Once deleted, the critical-section object cannot be specified in the **EnterCriticalSection** or **LeaveCriticalSection** function.

See Also

EnterCriticalSection
InitializeCriticalSection
LeaveCriticalSection

DeleteFile

DeleteFile deletes an existing file.

```
BOOL
DeleteFile(
    LPCTSTR lpFileName
    );
```

Parameters

lpFileName

Points to a null-terminated string that specifies the file to be deleted.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

If an application attempts to delete a file that does not exist, the **DeleteFile** function fails.

The **DeleteFile** function fails if an application attempts to delete a file that is open for normal I/O or as a memory-mapped file.

To close an open file, use the **CloseHandle** function.

See Also

CloseHandle
CreateFile

DeviceIoControl

DeviceIoControl sends a control code directly to a specified device driver, causing the corresponding device to perform the specified operation.

BOOL
DeviceIoControl(
HANDLE hDevice,
DWORD *IoControlCode*,
LPVOID *lpInBuffer*,
DWORD *nInBufferSize*,
LPVOID *lpOutBuffer*,
DWORD nOutBufferSize,
LPDWORD *lpBytesReturned*,
LPOVERLAPPED *lpOverlapped*
);

Parameters

hDevice

A handle to the device that is to perform the operation. Call **CreateFile** to obtain a device handle.

IoControlCode

The control code for the operation. This value identifies the specific operation to be performed and the type of device on which the operation is to be performed. The following values are defined:

IOCTL_DISK_CHECK_VERIFY
Obsolete. Use IOCTL_STORAGE_CHECK_VERIFY

IOCTL_DISK_EJECT_MEDIA
Obsolete. Use IOCTL_STORAGE_EJECT_MEDIA

IOCTL_DISK_FORMAT_TRACKS
Formats a contiguous set of disk tracks.

IOCTL_DISK_GET_DRIVE_GEOMETRY
Obtains information on the physical disk's geometry.

IOCTL_DISK_GET_DRIVE_LAYOUT
Provides information about each partition on a disk.

IOCTL_DISK_GET_MEDIA_TYPES
Obsolete. Use IOCTL_STORAGE_GET_MEDIA_TYPES

IOCTL_DISK_GET_PARTITION_INFO
Obtains disk partition information.

IOCTL_DISK_LOAD_MEDIA
Obsolete. Use IOCTL_STORAGE_LOAD_MEDIA

IOCTL_DISK_MEDIA_REMOVAL
Obsolete. Use IOCTL_STORAGE_MEDIA_REMOVAL

IOCTL_DISK_PERFORMANCE
Provides disk performance information.

IOCTL_DISK_REASSIGN_BLOCKS
Maps disk blocks to spare-block pool.

IOCTL_DISK_SET_DRIVE_LAYOUT
Partitions a disk.

IOCTL_DISK_SET_PARTITION_INFO
Sets the disk partition type.

IOCTL_DISK_VERIFY
Performs logical format of a disk extent.

IOCTL_SERIAL_LSRMST_INSERT
Enables or disables placement of a line and modem status data into the data stream.

IOCTL_STORAGE_CHECK_VERIFY
Checks for change in a removable-media device.

IOCTL_STORAGE_EJECT_MEDIA
Ejects media from a SCSI device.

IOCTL_STORAGE_GET_MEDIA_TYPES
Obtains information about media support.

IOCTL_STORAGE_LOAD_MEDIA
Loads media into a device.

IOCTL_STORAGE_MEDIA_REMOVAL
Enables or disables the media eject mechanism.

For more detailed information on each control code, see its topic in the Microsoft documentation. In particular, each topic provides details on the usage of the *lpInBuffer*, *nInBufferSize*, *lpOutBuffer*, *nOutBufferSize*, and *lpBytesReturned* parameters.

lpInBuffer

A pointer to a buffer that contains the data required to perform the operation.

This parameter can be NULL if the *IoControlCode* parameter specifies an operation that does not require input data.

nInBufferSize

The size, in bytes, of the buffer pointed to by lpInBuffer.

lpOutBuffer

A pointer to a buffer that receives the operation's output data.

This parameter can be NULL if the *IoControlCode* parameter specifies an operation that does not produce output data.

nOutBufferSize

The size, in bytes, of the buffer pointed to by *lpOutBuffer*.

lpBytesReturned

A pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by *lpOutBuffer*.

*lpOverlapped (**ignored by RTX**)*

This parameter should be set to NULL.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

CreateFile

DllMain

The **DllMain** function is an optional method of entry into a dynamic-link library (DLL). If the function is used, it is called by the system when processes and threads are initialized and terminated, or upon calls to **LoadLibrary** and **FreeLibrary**.

BOOL WINAPI
DllMain(
HINSTANCE *hinstDLL*,
DWORD *fdwReason*,
LPVOID *lpvReserved*
);

Parameters

HinstDLL

A handle to the DLL. HINSTANCE of a DLL is the same as the HMODULE of the DLL, so *hinstDLL* can be used in subsequent calls to functions that require a module handle.

fdwReason

Specifies a flag indicating why the DLL entry-point function is being called. This parameter can be one of the following values:

Value	Meaning
DLL_PROCESS_ATTACH	Indicates that the DLL is being loaded into the virtual address space of the current process as a result of a call to **LoadLibrary**.
DLL_THREAD_ATTACH	This value is not used in the RTSS Environment.
DLL_THREAD_DETACH	This value is not used in the RTSS Environment.
DLL_PROCESS_DETACH	Indicates that the DLL is being unloaded from the virtual address space of the calling process as a result of either a process exit or a call to **FreeLibrary**.

lpvReserved

lpvReserved is NULL.

Return Values

When the system calls **DllMain** with the DLL_PROCESS_ATTACH value, the function returns TRUE if it succeeds or FALSE if initialization fails. If the return value is FALSE, **LoadLibrary** returns NULL. To get extended error information, call **GetLastError**.

When the system calls **DllMain** with any value other than DLL_PROCESS_ATTACH, the return value is ignored.

Comments

RTSS calls **DllMain** only for thread-issuing **LoadLibrary** calls.

On attach, the body of the DLL entry-point function should perform only simple initialization tasks such as creating synchronization objects, and opening files. Do not call **LoadLibrary** in the entry-point function, because this may create dependency loops in the DLL load order. This can result in a DLL being used before the system has executed its initialization code. Similarly, do not call **FreeLibrary** in the entry-point function on detach because this can result in a DLL being used after the system has executed its termination code.

Calling Win32 functions other than synchronization, and file functions may result in problems that are difficult to diagnose. For example, calling User, Shell, COM, RPC, and Windows Sockets functions (or any functions that call these functions) can cause access violation errors because their DLLs call **LoadLibrary** to load other system components.

To provide more complex initialization, create an initialization routine for the DLL and require applications to call the initialization routine before calling any other routines in the DLL. Otherwise, have the initialization routine create a named mutex, and have each routine in the DLL call the initialization routine if the mutex does not exist.

EnterCriticalSection

EnterCriticalSection waits for ownership of the specified critical-section object. The function returns when the calling thread is granted ownership.

VOID
EnterCriticalSection(
LPCRITICAL_SECTION *lpCriticalSection*
);

Parameters

lpCriticalSection

A pointer to the critical-section object.

Return Values

This function does not return a value.

Comments

The threads of a single process can use a critical-section object for mutual-exclusion synchronization. The process is responsible for allocating the memory used by a critical-section object, which it can do by declaring a variable of type CRITICAL_SECTION. Before using a critical-section, some thread of the process must call the **InitializeCriticalSection** function to initialize the object.

To enable mutually exclusive access to a shared resource, each thread calls the **EnterCriticalSection** function to request ownership of the critical-section before executing any section of code that accesses the protected resource. **EnterCriticalSection** blocks until the thread can take ownership of the critical-section. When it has finished executing the protected code, the thread uses the **LeaveCriticalSection** function to relinquish ownership, enabling another thread to become owner and access the protected resource. The thread must call **LeaveCriticalSection** once for each time that it entered the critical-section. The thread enters the critical-section each time **EnterCriticalSection** succeeds.

Once a thread has ownership of a critical-section, it can make additional calls to **EnterCriticalSection** without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical-section that it already owns.

Any thread of the process can use the **DeleteCriticalSection** function to release the system resources that were allocated when the critical-section object was initialized.

After this function has been called, the critical-section object can no longer be used for synchronization.

See Also

DeleteCriticalSection, InitializeCriticalSection, LeaveCriticalSection

ExitProcess

ExitProcess ends a process and all its threads.

```
VOID
ExitProcess(
    UINT uExitCode
    );
```

Parameters

uExitCode (Ignored in RTSS)

The exit code for the process.

Return Values

This function does not return a value.

Comments

ExitProcess, **ExitThread**, **CreateThread**, and a process that is starting are serialized between each other within a process. Only one of these events can occur in an address space at a time.

See Also

CreateThread

ExitThread

GetExitCodeThread

ExitThread

ExitThread ends a thread.

VOID
ExitThread(
    DWORD *ExitCode*
    );

Parameters

ExitCode

The exit code for the calling thread. Use **GetExitCodeThread** to retrieve a thread's exit code.

Return Values

This function does not return a value.

Comments

ExitThread is the preferred method of exiting a thread. When this function is called (either explicitly or by returning from a thread procedure), the current thread's stack is de-allocated and the thread terminates.

If the thread is the last thread in the process when this function is called, the thread's process is also terminated.

Terminating a thread does not necessarily remove the thread object from the operating system. A thread object is deleted when the last handle to the thread is closed.

ExitProcess, **ExitThread**, **CreateThread**, and a process that is starting are serialized between each other within a process. Only one of these events can occur at a time.

If the primary thread calls **ExitThread**, the application will exit.

See Also

CreateThread
ExitProcess
GetExitCodeThread
TerminateThread

FreeLibrary

FreeLibrary decrements the reference count of the loaded dynamic-link library (DLL) module. When the reference count reaches zero, the module is unmapped from the address space of the calling process and the handle is no longer valid.

```
BOOL
FreeLibrary(
    HMODULE hLibModule
    );
```

Parameters

hLibModule

Handle to the loaded library module. The **LoadLibrary** function returns this handle.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

Each process maintains a reference count for each loaded library module.

This reference count is incremented each time **LoadLibrary** is called and is decremented each time **FreeLibrary** is called. A DLL module loaded at process initialization due to load-time dynamic linking has a reference count of one. This count is incremented if the same module is loaded by a call to **LoadLibrary**.

Before unmapping a library module, the system enables the DLL to detach from the process by calling the DLL's **DllMain** function, if it has one, with the DLL_PROCESS_DETACH value. Doing so gives the DLL an opportunity to clean up resources allocated on behalf of the current process. After the entry-point function returns, the library module is removed from the address space of the current process.

It is not safe to call **FreeLibrary** from **DllMain**. For more information, see the Comments section in **DllMain**.

Calling **FreeLibrary** does not affect other processes using the same library module.

See Also

DllMain
ExitThread
LoadLibrary

GetCurrentProcessId

GetCurrentProcessId returns the process identifier of the calling process.

DWORD
GetCurrentProcessId(VOID)

Parameters

This function has no parameters.

Return Values

The return value is the process identifier of the calling process.

Comments

Until the process terminates, the process identifier uniquely identifies the process throughout the system.

RTSSRun returns the value of the current process ID. In the RTSS environment, this is equivalent to the RTSS slot number.

See Also

RtOpenProcess

GetCurrentThread

GetCurrentThread returns a pseudohandle for the current thread.

HANDLE
GetCurrentThread(VOID)

Parameters

This function has no parameters.

Return Values

The return value is a pseudohandle for the current thread.

Comments

A pseudohandle is a special constant that is interpreted as the current thread handle. The calling thread can use this handle to specify itself whenever a thread handle is required. Pseudohandles are not inherited by child processes.

This handle has the maximum possible access to the thread object.

The function cannot be used by one thread to create a handle that can be used by other threads to refer to the first thread. The handle is always interpreted as referring to the thread that is using it.

See Also

CloseHandle
GetCurrentThreadId

GetCurrentThreadId

GetCurrentThreadId returns the thread identifier of the calling thread.

DWORD
GetCurrentThreadId(VOID)

Parameters

This function has no parameters.

Return Values

The return value is the thread identifier of the calling thread.

Comments

Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system.

See Also

GetCurrentThread

GetExceptionCode

GetExceptionCode retrieves a code that identifies the type of exception that occurred. The function can be called only from within the filter expression or exception-handler block of a try-except exception handler.

DWORD
GetExceptionCode(VOID)

Parameters

This function has no parameters.

Return Values

The return value identifies the type of exception. The following list shows the exception codes that are likely to occur due to common programming errors. For more information, see **RTX Exception Handling** in the *RTX User's Guide*.

Value	Meaning
EXCEPTION_ACCESS_VIOLATION	The thread attempted to read from or write to a virtual address for which it does not have the appropriate access.
EXCEPTION_BREAKPOINT	A breakpoint was encountered.
EXCEPTION_DATATYPE_MISALIGNMENT	The thread attempted to read or write data that is misaligned on hardware that does not provide alignment. For example, 16-bit values must be aligned on 2-byte boundaries, 32-bit values on 4-byte boundaries, and so on.
EXCEPTION_FLT_DENORMAL_OPERAND	One of the operands in a floating-point operation is denormal. A denormal value is one that is too small to represent as a standard floating-point value.
EXCEPTION_FLT_DIVIDE_BY_ZERO	The thread attempted to divide a floating-point value by a floating-point divisor of zero.
EXCEPTION_FLT_INEXACT_RESULT	The result of a floating-point operation cannot be represented exactly as a decimal fraction.
EXCEPTION_FLT_INVALID_OPERATION	This exception represents any floating-point exception not included in this list.
EXCEPTION_FLT_OVERFLOW	The exponent of a floating-point operation is greater than the magnitude allowed by the corresponding type.
EXCEPTION_FLT_UNDERFLOW	The exponent of a floating-point operation is less than the magnitude allowed by the corresponding type.
EXCEPTION_INT_DIVIDE_BY_ZERO	The thread attempted to divide an integer value by an integer divisor of zero.
EXCEPTION_ILLEGAL_INSTRUCTION	The method has terminated due to invalid parameters or property values.

Comments

GetExceptionCode can be called only from within the filter expression or exception-handler block of a try-except statement. The filter expression is evaluated if an exception occurs during execution of the try block, and it determines whether the except block is executed. The following example shows the structure of a try-except statement.

```
try {
/* try block */
}
except (filter-expression) {
/* exception handler block */
}
```

The filter expression can invoke a filter function. The filter function cannot call **GetExceptionCode**. However, the return value of **GetExceptionCode** can be passed as a parameter to a filter function. The return value of the **GetExceptionInformation** function can also be passed as a parameter to a filter function. **GetExceptionInformation** returns a pointer to a structure that includes the exception-code information. In the case of nested try-except statements, each statement's filter expression is evaluated until one is evaluated as EXCEPTION_EXECUTE_HANDLER or EXCEPTION_CONTINUE_EXECUTION. Each filter expression can invoke **GetExceptionCode** to get the exception code. The exception code returned is the code generated by a hardware exception, or the code specified in the **RaiseException** function for a software-generated exception.

See Also

GetExceptionInformation
RaiseException

GetExceptionInformation

GetExceptionInformation retrieves a machine-independent description of an exception, and information about the machine state that existed for the thread when the exception occurred. This function can be called only from within the filter expression of a try-except exception handler.

LPEXCEPTION_POINTERS
GetExceptionInformation(VOID)

Parameters

This function has no parameters.

Return Values

The return value is a pointer to an EXCEPTION_POINTERS structure that contains pointers to two other structures: an EXCEPTION_RECORD structure containing a description of the exception, and a CONTEXT structure containing the machine-state information.

Comments

The filter expression (from which the function is called) is evaluated if an exception occurs during execution of the try block, and it determines whether the except block is executed. The following example shows the structure of a try-except statement.

```
try {
/* try block */
}
except (filter-expression) {
/* exception handler block */
}
```

The filter expression can invoke a filter function. The filter function cannot call **GetExceptionInformation**. However, the return value of **GetExceptionInformation** can be passed as a parameter to a filter function.

To pass the EXCEPTION_POINTERS information to the exception-handler block, the filter expression or filter function must copy the pointer or the data to safe storage that the handler can later access. In the case of nested try-except statements, each statement's filter expression is evaluated until one is evaluated as EXCEPTION_EXECUTE_HANDLER or EXCEPTION_CONTINUE_EXECUTION. Each filter expression can invoke **GetExceptionInformation** to get exception information.

See Also

GetExceptionCode

GetExitCodeThread

GetExitCodeThread retrieves the termination status of the specified thread.

```
BOOL
GetExitCodeThread(
    HANDLE hThread,
    LPDWORD lpExitCode
    );
```

Parameters

hThread

The thread identifier.

lpExitCode

A pointer to a 32-bit variable to receive the thread termination status.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

If the specified thread has not terminated, the termination status returned is STILL_ACTIVE. If the thread has terminated, the termination status returned may be one of the following:

- The exit value specified in **ExitThread** or **TerminateThread**
- The return value from the thread function
- The exit value of the thread's process

See Also

ExitThread
TerminateThread

GetLastError

GetLastError returns the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code.

DWORD
GetLastError(VOID)

Parameters

This function has no parameters.

Return Values

The return value is the calling thread's last-error code value. Functions set this value by calling **SetLastError**.

Comments

Call **GetLastError** immediately when a function's return value indicates that such a call will return useful data. That is because some functions call **SetLastError(0)** when they succeed, wiping out the error code set by the most recently failed function.

Most functions provided in RTX that set the thread's last error code value set it when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, 0XFFFFFFFF, or -1. Some functions call **SetLastError** under conditions of success; those cases are noted in each function's reference page.

GetProcAddress

GetProcAddress returns the address of the specified exported dynamic-link library (DLL) function.

FARPROC
GetProcAddress(
HMODULE *hModule*,
LPCSTR *lpProcName*
);

Parameters

hModule

A handle to the DLL module that contains the function. The **LoadLibrary** function returns this handle.

lpProcName

A pointer to a null-terminated string containing the function name, or the function's ordinal value.

Return Values

If the function succeeds, the return value is the address of the DLL's exported function.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

GetProcAddress is used to retrieve addresses of exported functions in DLLs.

The spelling and case of the function name pointed to by *lpProcName* must be identical to that in the EXPORTS statement of the source DLL's module-definition (.DEF) file. The exported names of Win32 API functions may differ from the names used when calling these functions in the code. This difference is hidden by macros used in the SDK header files. For more information, see Win32 Function Prototypes.

The *lpProcName* parameter can identify the DLL function by specifying an ordinal value associated with the function in the EXPORTS statement.

GetProcAddress verifies that the specified ordinal is in the range 1 through the highest ordinal value exported in the .DEF file. The function then uses the ordinal as an index to read the function's address from a function table. If the .DEF file does not number the functions consecutively from 1 to N (where N is the number of exported functions), an error can occur where **GetProcAddress** returns an invalid, non-NULL address, even though there is no function with the specified ordinal.

In cases where the function may not exist, the function should be specified by name rather than by ordinal value.

RTSS Environment: The following information applies to the RTSS environment:

- Function lookup by ordinal value is not presently supported.
- If the exported routine name is decorated (e.g., _<fname>@<# argument bytes> as in the __stdcall convention) the decorated name must be specified in the call to **GetProcAddress**.
- DEF files are not supported.

See Also

FreeLibrary
LoadLibrary

GetProcessHeap

GetProcessHeap obtains a handle to the heap of the calling process. This handle can then be used in calls to **HeapAlloc**, **HeapReAlloc**, **HeapFree**, and **HeapSize**.

HANDLE
GetProcessHeap(VOID)

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a handle to the calling process's heap.

If the function fails, the return value is NULL.

Comments

GetProcessHeap allows RTSS processes to allocate memory from the process heap. The following example shows how to use this call with **HeapAlloc**.

```
HeapAlloc(GetProcessHeap(), 0, dwBytes);
```

See Also

HeapAlloc
HeapFree
HeapReAlloc
HeapSize

GetThreadPriority

GetThreadPriority returns the priority value for the specified thread. This value, together with the priority class of the thread's process, determines the thread's base-priority level.

INT
GetThreadPriority(
HANDLE *hThread*
);

Parameters

hThread

The thread identifier.

Return Values

If the function succeeds, the return value is the thread's priority level.

If the function fails, the return value is THREAD_PRIORITY_ERROR_RETURN. To get extended error information, call **GetLastError**.

Comments

See the Comments section in **RtGetThreadPriority** for details on thread priority mapping.

See Also

CreateThread
RtGetThreadPriority
RtSetThreadPriority
SetThreadPriority

HeapAlloc

HeapAlloc allocates a block of memory from a heap. The allocated memory is not movable.

```
LPVOID
HeapAlloc(
    HANDLE hHeap,
    DWORD Flags,
    DWORD Bytes
    );
```

Parameters

hHeap

The heap from which the memory will be allocated. This parameter is a handle returned by **GetProcessHeap**.

Flags

The controllable aspects of heap allocation. You can specify the following flag:

HEAP_ZERO_MEMORY
The allocated memory will be initialized to zero.

Bytes

The number of bytes to be allocated.

Return Values

If the function succeeds, the return value is a pointer to the allocated memory block.

If the function fails, the return value is NULL.

Comments

If **HeapAlloc** succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, because it is rounded to page boundry, the process can use the entire amount. To determine the actual size of the allocated block, use **HeapSize**.

To free a block of memory allocated by **HeapAlloc**, use **HeapFree**. Memory allocated by **HeapAlloc** is not movable. Since the memory is not movable, it is possible for the heap to become fragmented. Note that if HEAP_ZERO_MEMORY is not specified, the allocated memory may not be initialized to zero.

See Also

GetProcessHeap, HeapFree, HeapReAlloc, HeapSize, SetLastError

HeapCreate

HeapCreate creates a heap object that can be used by the calling process. The function reserves a contiguous block in the virtual address space of the process and allocates physical storage for a specified initial portion of this block.

```
HANDLE
HeapCreate(
    DWORD flOptions,
    DWORD InitialSize,
    DWORD MaximumSize
    );
```

Parameters

flOptions

The optional attributes for the new heap. These flags will affect subsequent access to the new heap through calls to the heap functions (**HeapAlloc**, **HeapFree**, **HeapReAlloc**, and **HeapSize**).

You can specify one or more of the following flags:

HEAP_GENERATE_EXCEPTIONS
Specifies that the system will raise an exception to indicate a function failure, such as an out-of-memory condition, instead of returning NULL.

HEAP_NO_SERIALIZE
Specifies that mutual exclusion will not be used when the heap functions allocate and free memory from this heap. The default, occurring when the HEAP_NO_SERIALIZE flag is not specified, is to serialize access to the heap. Serialization of heap access allows two or more threads to simultaneously allocate and free memory from the same heap.

InitialSize

The initial size, in bytes, of the heap. This value determines the initial amount of physical storage that is allocated for the heap. The value is rounded up to the next page boundary.

MaximumSize

If *MaximumSize* is a non-zero value, it specifies the maximum size, in bytes, of the heap. **HeapCreate** rounds *MaximumSize* up to the next page boundary, and then reserves a block of that size in the process's virtual address space for the heap. If allocation requests made by **HeapAlloc** or **HeapReAlloc** exceed the initial amount of physical storage specified by *InitialSize*, the system allocates additional pages of physical storage for the heap, up to the heap's maximum size.

If *MaximumSize* is non-zero, the heap cannot grow, and an absolute limitation arises: the maximum size of a memory block in the heap is a bit less than 0x7FFF8 bytes. Requests to allocate larger blocks will fail, even if the maximum size of the heap is large enough to contain the block.

If *MaximumSize* is zero, it specifies that the heap can grow. The heap's size is limited only by available memory. Requests to allocate blocks larger than 0x7FFF8 bytes do not automatically fail; the system calls **VirtualAlloc** to obtain the memory needed for such large blocks. Applications that need to allocate large memory blocks should set *MaximumSize* to zero.

Return Values

If the function succeeds, the return value is a handle of the newly created heap.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

HeapCreate creates a private heap object from which the calling process can allocate memory blocks by using **HeapAlloc**. The initial size determines the number of committed pages that are initially allocated for the heap. The maximum size determines the total number of reserved pages. These pages create a contiguous block in the process's virtual address space into which the heap can grow. If requests by HeapAlloc exceed the current size of committed pages, additional pages are automatically committed from this reserved space, assuming that the physical storage is available.

The memory of a private heap object is accessible only to the process that created it. If a dynamic-link library (DLL) creates a private heap, the heap is created in the address space of the process that called the DLL, and it is accessible only to that process.

The system uses memory from the private heap to store heap support structures, so not all of the specified heap size is available to the process. For example, if **HeapAlloc** requests 64 Kilobytes (K) from a heap with a maximum size of 64K, the request may fail because of system overhead.

If the HEAP_NO_SERIALIZE flag is not specified (the simple default), the heap will serialize access within the calling process. Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap.

Setting the HEAP_NO_SERIALIZE flag eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE flag can, therefore, be safely used only in the following situations:

- The process has only one thread.
- The process has multiple threads, but only one thread calls the heap functions for a specific heap.
- The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

See Also

GetProcessHeap, HeapAlloc, HeapDestroy, HeapFree,HeapReAlloc, HeapSize

HeapDestroy

HeapDestroy destroys the specified heap object. It uncommits and releases all the pages of a private heap object and it invalidates the handle of the heap.

BOOL
HeapDestroy(
HANDLE *hHeap*
);

Parameters

hHeap

The heap to be destroyed. This parameter should be a heap handle returned by **HeapCreate**. A heap handle returned by **GetProcessHeap** should not be used.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

Processes can call **HeapDestroy** without first calling **HeapFree** to free memory allocated from the heap.

See Also

GetProcessHeap
HeapAlloc
HeapCreate
HeapFree
HeapReAlloc
HeapSize

HeapFree

HeapFree frees a memory block allocated from a heap by **HeapAlloc** or **HeapReAlloc**.

```
BOOL
HeapFree(
    HANDLE hHeap,
    DWORD Flags,
    LPVOID lpMem
    );
```

Parameters

hHeap

The heap whose memory block the function frees. This parameter is the handle returned by **GetProcessHeap**.

Flags (ignored)

lpMem

A pointer to the memory block to free. This pointer is returned by **HeapAlloc** or **HeapReAlloc**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

GetProcessHeap
HeapAlloc
HeapReAlloc
HeapSize
SetLastError

HeapReAlloc

HeapReAlloc reallocates a block of memory from a heap. This function enables you to resize a memory block and change other memory block properties.

LPVOID
HeapReAlloc(
HANDLE *hHeap*,
DWORD *Flags*,
LPVOID *lpMem*,
DWORD *Bytes*
);

Parameters

hHeap

The heap from which the memory will be reallocated. This is the handle returned by **GetProcessHeap**.

Flags

The controllable aspects of heap reallocation. You can specify one or both of the following flags:

HEAP_REALLOC_IN_PLACE_ONLY (not supported in RTSS)
Specifies that there can be no movement when reallocating a memory block to a larger size. If this flag is not specified and the reallocation request is for a larger size, the function may move the block to a new location. If this flag is specified and the block cannot be enlarged without moving, the function will fail, leaving the original memory block unchanged. Because memory movement always occurs, this flag is not supported in the RTSS environment.

HEAP_ZERO_MEMORY
If the reallocation request is for a larger size, this flag specifies that the additional region of memory beyond the original size will be initialized to zero. The contents of the memory block—up to its original size—are unaffected.

lpMem

A pointer to the block of memory that the function reallocates. This pointer is returned by an earlier call to **HeapAlloc** or **HeapReAlloc**.

Bytes

The new size of the memory block, in bytes. A memory block's size can be increased or decreased by using this function.

Return Values

If the function succeeds, the return value is a pointer to the reallocated memory block.

If the function fails, the return value is NULL. It calls **SetLastError**. To get extended error

information, call **GetLastError**.

Comments

If **HeapReAlloc** succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual size of the reallocated block, use **HeapSize**.

To free a block of memory allocated by HeapReAlloc, use **HeapFree**.

See Also

GetProcessHeap
HeapAlloc
HeapFree
HeapSize
SetLastError

HeapSize

HeapSize returns the size, in bytes, of a memory block allocated from a heap by **HeapAlloc** or **HeapReAlloc**.

```
DWORD
HeapSize(
    HANDLE hHeap,
    DWORD Flags,
    LPCVOID lpMem
    );
```

Parameters

hHeap

The heap in which the memory block resides. This handle is returned by **GetProcessHeap**.

Flags (ignored)

lpMem

A pointer to the memory block whose size the function will obtain. This pointer is returned by **HeapAlloc** or **HeapReAlloc**.

Return Values

If the function succeeds, the return value is the size, in bytes, of the allocated memory block.

If the function fails, the return value is 0XFFFFFFFF. The function does not call **SetLastError**. An application cannot call **GetLastError** for extended error information.

See Also

GetProcessHeap
HeapAlloc
HeapFree
HeapReAlloc
SetLastError

InitializeCriticalSection

InitializeCriticalSection initializes a critical section object.

VOID
InitializeCriticalSection(
LPCRITICAL_SECTION *lpCriticalSection*
);

Parameters

lpCriticalSection

A pointer to the critical section object.

Return Values

This function does not return a value.

Comments

The threads of a single process can use a critical section object for mutual-exclusion synchronization. The process is responsible for allocating the memory used by a critical section object, which it can do by declaring a variable of type CRITICAL_SECTION. Before using a critical section, some thread of the process must call **InitializeCriticalSection** to initialize the object.

Once a critical-section object has been initialized, the threads of the process can specify the object in **EnterCriticalSection** or **LeaveCriticalSection** to provide mutually exclusive access to a shared resource. For similar synchronization between the threads of different processes, use a mutex object.

A critical-section object cannot be moved or copied. The process must also not modify the object, but must treat it as logically opaque.

See Also

DeleteCriticalSection
EnterCriticalSection
LeaveCriticalSection
RtCreateMutex

LeaveCriticalSection

LeaveCriticalSection releases ownership of the specified critical-section object.

```
VOID
LeaveCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
    );
```

Parameters

lpCriticalSection

A pointer to the critical-section object.

Return Values

This function does not return a value.

Comments

The threads of a single process can use a critical-section object for mutual-exclusion synchronization. The process is responsible for allocating the memory used by a critical-section object, which it can do by declaring a variable of type CRITICAL_SECTION.

Before using a critical-section, some thread of the process must call the **InitializeCriticalSection** function to initialize the object. A thread uses the **EnterCriticalSection** function to acquire ownership of a critical-section object. To release its ownership, the thread must call **LeaveCriticalSection** once for each time that it entered the critical-section. If a thread calls **LeaveCriticalSection** when it does not have ownership of the specified critical-section object, an error occurs that may cause another thread using **EnterCriticalSection** to wait indefinitely. Any thread of the process can use the **DeleteCriticalSection** function to release the system resources that were allocated when the critical-section object was initialized. After this function has been called, the critical-section object can no longer be used for synchronization.

See Also

DeleteCriticalSection
EnterCriticalSection
InitializeCriticalSection

LoadLibrary

LoadLibrary maps the specified executable module into the address space of the calling process.

HPINSTANCE
LoadLibrary(
LPCTSTR *lpLibFileName*
);

Parameters

LpLibFileName

Pointer to a null-terminated string that names the executable module (either a DLL or EXE file). The name specified is the filename of the module and is not related to the name stored in the library module itself, as specified by the library keyword in the module definition (DEF) file.

If the string specifies a path but the path does not exist in the specified directory, the function fails.

If the string does not specify a path, the function uses a standard search strategy to find the file.

For RTDLLs, *lplibfilename* should have the extension .dll, not .rtdll. A path is not necessary to provide, but the RTDLL should be registered.

Return Values

If the function succeeds, the return value is a handle to the module.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Comments

LoadLibrary can be used to map a DLL module and return a handle that can be used in **GetProcAddress** to get the address of a DLL function. **LoadLibrary** can also be used to map other executable modules. For example, the function can specify an EXE file to get a handle that can be used in **FindResource** or **LoadResource**.

Note: Do not use **LoadLibrary** to run an EXE file.

If the module's DLL is not already mapped for the calling process, the system calls the DLL's **DllMain** function with the DLL_PROCESS_ATTACH value.

If the DLL's entry-point function does not return TRUE, **LoadLibrary** fails and returns NULL.

Note: It is not safe to call **LoadLibrary** from **DllMain**.

Module handles are not global or inheritable. A call to **LoadLibrary** by one process does not

produce a handle that another process can use, for example, in calling **GetProcAddress**. The other process must make its own call to **LoadLibrary** for the module before calling **GetProcAddress**.

If no filename extension is specified in the *lpLibFileName* parameter, the default library extension .DLL is appended. However, the filename string can include a trailing point character (.) to indicate that the module name has no extension. When no path is specified, the function searches for loaded modules whose base name matches the base name of the module to be loaded. If the name matches, the load succeeds. Otherwise, the function searches for the file in the following sequence:

1. The directory from which the application loaded.
2. The current directory.
3. The 32-bit Windows system directory. (Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.)
4. The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM. Use the **GetWindowsDirectory** function to get the path of this directory. The directories that are listed in the PATH environment variable. The first directory searched is the one directory containing the image file used to create the calling process (for more information, see **CreateProcess** in the Microsoft SDK documentation). Doing this allows private dynamic-link library (DLL) files associated with a process to be found without adding the process's installed directory to the PATH environment variable.

The Visual C++ compiler supports a syntax that enables you to declare thread-local variables: _declspec(thread). If you use this syntax in a DLL, you will not be able to load the DLL explicitly using **LoadLibrary**. If your DLL will be loaded explicitly, you must use the thread local storage functions instead of _declspec(thread).

RTSS Environment: The following information applies to the RTSS environment.

- The .exe extension and DLL names without an extension (i.e., the trailing dot convention) are not supported.
- Any path specified as part of *lpLibFileName* is ignored. RTDLLs are loaded based on whether the filename specified matches an existing, registered RTDLL.
- DLLs must be registered through "RTSSRun /dll <image name>" before they can be successfully accessed with **LoadLibrary**.
- _declspec(thread) is not supported.

See Also

DllMain
GetProcAddress

RaiseException

RaiseException raises an exception in the calling thread.

```
VOID
RaiseException(
    DWORD ExceptionCode,
    DWORD ExceptionFlags,
    DWORD nNumberOfArguments,
    CONST DWORD* lpArguments
    );
```

Parameters

ExceptionCode

The application-defined exception code of the exception being raised. The filter expression and exception-handler block of an exception handler can use **GetExceptionCode** to retrieve this value.

Note that the system will clear bit 28 of *ExceptionCode*. This bit is a reserved exception bit, used by the system for its own purposes. For example, after calling **RaiseException** with an *ExceptionCode* value of 0XFFFFFFFF, Windows displays a message indicating that the exception number is 0XEFFFFFFF.

ExceptionFlags

The exception flags. This can be either zero to indicate a continuable exception, or EXCEPTION_NONCONTINUABLE to indicate a non-continuable exception.

A non-continuable exception causes the process to unload or freeze with a "Non-continuable Exception" message to prevent a stack fault in RTSS.

Note: The Win32 behavior differs; it continues re-raising EXCEPTION_NONCONTINUABLE_EXCEPTION.

nNumberOfArguments

The number of arguments in the *lpArguments* array. This value must not exceed EXCEPTION_MAXIMUM_PARAMETERS. This parameter is ignored if *lpArguments* is NULL.

lpArguments

A pointer to an array of 32-bit arguments. This parameter can be NULL. These arguments can contain any application-defined data that needs to be passed to the filter expression of the exception handler.

Return Values

This function does not return a value.

Comments

RaiseException enables a process to use structured exception handling to handle private, software-generated, application-defined exceptions. Raising an exception causes the exception dispatcher to go through the following search for an exception handler:

1. The system attempts to locate a frame-based exception handler by searching the stack frames of the thread in which the exception occurred. The system searches the current stack frame first, then proceeds backward through preceding stack frames.
2. If no frame-based handler can be found, or no frame-based handler handles the exception, the system provides default handling based on the exception type. For most exceptions, the default action is to call **ExitProcess**.

The values specified in the *ExceptionCode*, *ExceptionFlags*, *nNumberOfArguments*, and *lpArguments* parameters can be retrieved in the filter expression of a try-except frame-based exception handler by calling **GetExceptionInformation**.

See Also

ExitProcess
GetExceptionCode
GetExceptionInformation

ReadFile

ReadFile reads data from a file, starting at the position indicated by the file pointer. After the read operation has been completed, the file pointer is adjusted by the number of bytes actually read.

```
BOOL
ReadFile(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped
    );
```

Parameters

hFile

The file to be read. The file handle must have been created with GENERIC_READ access to the file.

lpBuffer

A pointer to the buffer that receives the data read from the file.

nNumberOfBytesToRead

The number of bytes to be read from the file.

lpNumberOfBytesRead

A pointer to the number of bytes read. **ReadFile** sets this value to zero before doing any work or error checking. If this parameter is zero when **ReadFile** returns TRUE on a named pipe, the other end of the message-mode pipe called **WriteFile** with *nNumberOfBytesToWrite* set to zero.

lpOverlapped ***(not supported)***

This parameter must be set to NULL.

Return Values

If the function succeeds, the return value is TRUE.

If the return value is TRUE and the number of bytes read is zero, the file pointer was beyond the current end of the file at the time of the read operation.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

ReadFile returns when the number of bytes requested has been read, or an error occurs.

If part of the file is locked by another process and the read operation overlaps the locked portion, this function fails.

Applications must not read from nor write to the input buffer that a read operation is using until the read operation completes. A premature access to the input buffer may lead to corruption of the data read into that buffer.

When a synchronous read operation reaches the end of a file, **ReadFile** returns TRUE and sets **lpNumberOfBytesRead* to zero.

See Also

CreateFile
WriteFile

RemoveDirectory

RemoveDirectory deletes an existing empty directory.

```
BOOL
RemoveDirectory(
    LPCTSTR lpPathName
    );
```

Parameters

lpPathName

A pointer to a null-terminated string that specifies the path of the directory to be removed. The path must specify an empty directory, and the calling process must have delete access to the directory.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

RemoveDirectory does not delete any object identified by *lpPathName*.

See Also

CreateDirectory

ResumeThread

ResumeThread subtracts one from a thread's suspend count. When the suspend count is reduced to zero, the execution of the thread is resumed.

DWORD
ResumeThread(
HANDLE *hThread*
);

Parameters

hThread

A handle for the thread to be restarted.

Return Values

If the function succeeds, the return value is the thread's previous suspend count.

If the function fails, the return value is 0XFFFFFFFF. To get extended error information, call **GetLastError**.

Comments

ResumeThread checks the suspend count of the subject thread. If the suspend count is 0, the thread is not currently suspended. Otherwise, the subject thread's suspend count is reduced by one. If the resulting value is 0, then the execution of the subject thread is resumed.

If the return value is 0, the specified thread was not suspended. If the return value is 1, the specified thread was suspended but was restarted. If the return value is greater than 1, the specified thread is still suspended.

See Also

SuspendThread

SetFilePointer

SetFilePointer moves the file pointer of an open file.

```
DWORD
SetFilePointer(
    HANDLE hFile,
    LONG lDistanceToMove,
    PLONG lpDistanceToMoveHigh,
    DWORD MoveMethod
    );
```

Parameters

hFile

The file whose file pointer is to be moved. The file handle must have been created with GENERIC_READ or GENERIC_WRITE access to the file.

lDistanceToMove

The number of bytes to move the file pointer. A positive value moves the pointer forward in the file and a negative value moves it backward.

lpDistanceToMoveHigh

A pointer to the high-order word of the 64-bit distance to move. If the value of this parameter is NULL, **SetFilePointer** can operate only on files whose maximum size is 2^32 - 2. If this parameter is specified, the maximum file size is 2^64 - 2. This parameter also receives the high-order word of the new value of the file pointer.

MoveMethod

The starting point for the file pointer move. This parameter can be one of the following values:

Value	Meaning
FILE_BEGIN	The starting point is zero or the beginning of the file. If FILE_BEGIN is specified, *DistanceToMove* is interpreted as an unsigned location for the new file pointer.
FILE_CURRENT	The current value of the file pointer is the starting point.
FILE_END	The current end-of-file position is the starting point.

Return Values

If **SetFilePointer** succeeds, the return value is the low-order double-word of the new file pointer, and if *lpDistanceToMoveHigh* is not NULL, the function puts the high-order double-word of the new file pointer into the LONG pointed to by that parameter.

If the function fails and *lpDistanceToMoveHigh* is NULL, the return value is 0xFFFFFFFF. To get extended error information, call **GetLastError**.

If the function fails and *lpDistanceToMoveHigh* is non-NULL, the return value is 0xFFFFFFFF and **GetLastError** will return a value other than NO_ERROR.

Comments

Do not use **SetFilePointer** with a handle to a non-seeking device, such as a pipe or a communications device.

Use caution when setting the file pointer in a multithreaded application. For example, an application whose threads share a file handle, update the file pointer, and read from the file must protect this sequence by using a critical section object or mutex object.

If the hFile file handle was opened with the FILE_FLAG_NO_BUFFERING flag set, an application can move the file pointer only to sector-aligned positions. A sector-aligned position is a position that is a whole number multiple of the volume's sector size. If an application calls **SetFilePointer** with distance-to-move values that result in a position that is not sector-aligned and a handle that was opened with FILE_FLAG_NO_BUFFERING, the function fails, and **GetLastError** returns ERROR_INVALID_PARAMETER.

If the return value is 0xFFFFFFFF and *lpDistanceToMoveHigh* is non-NULL, an application must call **GetLastError** to determine whether the function has succeeded or failed.

```
  } // end of error handler


//
// Case Two: calling the function with
// lpDistanceToMoveHigh != NULL


// try to move hFile's file pointer some huge distance
dwPointerLow = SetFilePointer (hFile, lDistanceLow, & lDistanceHigh,
FILE_BEGIN) ;


// if we failed ...
if (dwPointerLow == 0xFFFFFFFF
  &&
  (dwError = GetLastError()) != NO_ERROR ){


  // deal with that failure
  .
  .
   } // end of error handler
```

See Also

ReadFile
WriteFile
InitializeCriticalSection
EnterCriticalSection
LeaveCriticalSection
DeleteCriticalSection
Mutex Objects

SetLastError

SetLastError sets the last-error code for the calling thread.

VOID
SetLastError(
DWORD *ErrCode*
);

Parameters

ErrCode

The last error code for the thread.

Return Values

This function does not return a value.

Comments

Error codes are 32-bit values. (Bit 31 is the most significant bit.) Bit 29 is reserved for application-defined error codes; no RTAPI error code has this bit set. If you are defining an error code for your application, set this bit to indicate that the error code has been defined by your application and ensure that your error code does not conflict with any system-defined error codes.

Most functions provided in RTX call **SetLastError** when they fail. Function failure is typically indicated by a return value error code such as FALSE, NULL, 0XFFFFFFFF, or -1.

Applications can retrieve the value saved by this function by using **GetLastError**. The use of **GetLastError** is optional; an application can call it to find out the specific reason for a function failure.

The last error code is kept in thread local storage so that multiple threads do not overwrite each other's values.

See Also

GetLastError

SetThreadPriority

SetThreadPriority sets the priority value for the specified thread.

```
BOOL
SetThreadPriority(
    HANDLE hThread,
    int nPriority
    );
```

Parameters

hThread

The thread whose priority value is to be set.

nPriority

See the Comments section in **RtGetThreadPriority** for details on thread mapping priority.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

GetThreadPriority
RtGetThreadPriority
RtSetThreadPriority

SetUnhandledExceptionFilter

SetUnhandledExceptionFilter lets an application supersede the top-level exception handler that RTSS places at the top of each thread and process. After calling this function, if an exception occurs in a process and the system's scan of handlers reaches the RTSS unhandled exception filter, that filter will call the exception filter function specified by the *lpTopLevelExceptionFilter* parameter.

LPTOP_LEVEL_EXCEPTION_FILTER
SetUnhandledExceptionFilter(
LPTOP_LEVEL_EXCEPTION_FILTER *pTopLevelExceptionFilter*
);

Parameters

lpTopLevelExceptionFilter

The address of a top-level exception filter function that will be called whenever the **UnhandledExceptionFilter** function gets control. A value of NULL for this parameter specifies default handling within **UnhandledExceptionFilter**. The filter function has syntax congruent to that of **UnhandledExceptionFilter**: It takes a single parameter of type LPEXCEPTION_POINTERS, and returns a value of type LONG. The filter function returns one of the following values:

Value	Meaning
EXCEPTION_EXECUTE_HANDLER	Return from **UnhandledExceptionFilter** and execute the associated exception handler. This usually results in process termination.
EXCEPTION_CONTINUE _EXECUTION	Return from **UnhandledExceptionFilter** and continue execution from the point of the exception. Note that the filter function is free to modify the continuation state by modifying the exception information supplied through its *lpException_Pointers* parameter.
EXCEPTION_CONTINUE_SEARCH	Proceed with normal execution of **UnhandledExceptionFilter**. On an exception, RTSS always displays an Application Error message box stating that the application has been frozen or unloaded. The Win32 **UnhandledExceptionFilter** semantics provide the option to disable the exception-related pop-up via the **SetErrorMode** function with the SEM_NOGPFAULTERRORBOX flag.

Return Values

SetUnhandledExceptionFilter returns the address of the previous exception filter established with the function. A NULL return value means there is no current top-level exception handler.

Comments

Issuing **SetUnhandledExceptionFilter** replaces the existing top-level exception filter for all existing and all future threads in the calling process.

The exception handler specified by *lpTopLevelExceptionFilter* is executed in the context of the thread that caused the fault. This can affect the exception handler's ability to recover from certain exceptions, such as an invalid stack.

See Also

UnhandledExceptionFilter

Sleep

Sleep suspends the current process for the specified time.

```
VOID
Sleep(
    ULONG milliSeconds
    );
```

Parameters

milliSeconds

The amount of time to sleep, expressed as milliseconds.

Return Values

The function returns no value.

Comments

Sleep suspends the given thread from execution for the specified amount of time.

See Also

RtCreateTimer
RtDeleteTimer
RtGetClockResolution
RtGetClockTime
RtGetClockTimerPeriod
RtGetTimer
RtSetClockTime
RtSetTimer
RtSetTimerRelative

SuspendThread

SuspendThread suspends the specified thread.

```
DWORD
SuspendThread(
    HANDLE hThread
    );
```

Parameters

hThread

The thread to suspend.

Return Values

If the function succeeds, the return value is the thread's previous suspend count; otherwise, it is 0XFFFFFFFF. To get extended error information, use **GetLastError**.

Comments

If the function succeeds, execution of the specified thread is suspended and the thread's suspend count is raised by one.

Suspending a thread causes the thread to stop executing.

Each thread has a suspend count (with a maximum value of MAXIMUM_SUSPEND_COUNT). If the suspend count is greater than zero, the thread is suspended; otherwise, the thread is not suspended and is eligible for execution. Calling **SuspendThread** causes the target thread's suspend count to be raised by one. Attempting to increment past the maximum suspend count causes an error without incrementing the count.

ResumeThread decrements the suspend count of a suspended thread.

See Also

ResumeThread

TerminateThread

TerminateThread terminates a thread.

BOOL
TerminateThread(
HANDLE *hThread*,
DWORD *ExitCode*
);

Parameters

hThread

The thread to terminate.

ExitCode

The exit code for the thread. Use **GetExitCodeThread** to retrieve a thread's exit value.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

TerminateThread is used to cause a thread to exit. When this occurs, the target thread has no chance to execute any user-mode code. **TerminateThread** is a **dangerous** function that should only be used in the most extreme cases. Call **TerminateThread** only if you know exactly what the target thread is doing, and you control all of the code that the target thread could possibly be running at the time of the termination.

A thread cannot protect itself against **TerminateThread**, other than by controlling access to its handles.

If the target thread is the last thread of a process when this function is called, the thread's process is also terminated.

Terminating a thread does not necessarily remove the thread object from the system. A thread object is deleted when the last thread handle is closed.

See Also

CreateThread
ExitThread
GetExitCodeThread

TlsAlloc

TlsAlloc allocates a thread local storage (TLS) index. Any thread of the process can subsequently use this index to store and retrieve values that are local to the thread.

DWORD
TlsAlloc(VOID)

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a TLS index.

If the function fails, the return value is 0xFFFFFFFF. To get extended errorinformation, call **GetLastError**.

Comments

The threads of the process can use the TLS index in subsequent calls to **TlsFree**, **TlsSetValue**,or **TlsGetValue**.

TLS indexes are typically allocated during process or dynamic-link library(DLL) initialization. Once allocated, each thread of the process can use a TLSindex to access its own TLS storage slot. To store a value in its slot, athread specifies the index in a call to **TlsSetValue**.The thread specifies the same index in a subsequent call to **TlsGetValue**, to retrieve the stored value.

The constant TLS_MINIMUM_AVAILABLE defines the minimum number of TLS indexesavailable in each process. This minimum is guaranteed to be at least 64 for allsystems.

TLS indexes are not valid across process boundaries. A DLL cannot assumethat an index assigned in one process is valid in another process. A DLL mightuse **TlsAlloc**, **TlsSetValue**,**TlsGetValue**, and **TlsFree**as follows:

- When a DLL attaches to a process, the DLL uses **TlsAlloc**to allocate a TLS index. The DLL then allocates some dynamic storage and usesthe TLS index in a call to **TlsSetValue** tostore the address in the TLS slot. This concludes the per-thread initializationfor the initial thread of the process. The TLS index is stored in a global orstatic variable of the DLL.
- Each time the DLL attaches to a new thread of theprocess, the DLL allocates some dynamic storage for the new thread and uses theTLS index in a call to **TlsSetValue** tostore the address in the TLS slot. This concludes the per-thread initializationfor the new thread.
- Each time an initialized thread makes a DLL callrequiring the data in its dynamic storage, the DLL uses the TLS index in a callto **TlsGetValue** to retrieve the address ofthe dynamic storage for that thread.

Note: Since DllMain is only called for RTDLLs atprocess attach (and not thread attach) it can only be used in an RTDLL to maintainper-process data for the initializing thread. It can not

be used in an RTDLL to maintain thread local storage for anyadditional threads.

See Also

TlsFree
TlsGetValue
TlsSetValue

TlsFree

TlsFree releases a thread local storage (TLS) index, making it available for reuse.

BOOL
TlsFree(
DWORD *TlsIndex*
);

Parameters

TlsIndex

The TLS index that was allocated by **TlsAlloc**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

If the threads of the process have allocated dynamic storage and used the TLS index to store pointers to this storage, they should free the storage before calling TlsFree. The **TlsFree** function does not free any dynamic storage that has been associated with the TLS index. It is expected that DLLs call this function (if at all) only during their process detach routine.

For a brief discussion of typical uses of the TLS functions, see the Comments section of **TlsAlloc**.

See Also

TlsAlloc
TlsGetValue
TlsSetValue

TlsGetValue

TlsGetValue retrieves the value in the calling thread's thread local storage (TLS) slot for a specified TLS index. Each thread of a process has its own slot for each TLS index.

LPVOID
TlsGetValue(
DWORD *TlsIndex*
);

Parameters

TlsIndex

The TLS index that was allocated by **TlsAlloc**.

Return Values

If the function succeeds, the return value is the value stored in the calling thread's TLS slot associated with the specified index.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Note that the data stored in a TLS slot can have a value of zero. In this case, the return value is zero and **GetLastError** returns NO_ERROR.

Comments

TLS indexes are typically allocated by **TlsAlloc** during process or DLL initialization. Once allocated, each thread of the process can use a TLS index to access its own TLS storage slot for that index. The storage slot for each thread is initialized to NULL. A thread specifies a TLS index in a call to **TlsSetValue**, to store a value in its slot. The thread specifies the same index in a subsequent call to **TlsGetValue**, to retrieve the stored value.

TlsSetValue and **TlsGetValue** were implemented with speed as the primary goal. These functions perform minimal parameter validation and error checking. In particular, this function succeeds if *TlsIndex* is in the range 0 through (TLS_MINIMUM_AVAILABLE - 1). It is up to the programmer to ensure that the index is valid.

Win32 functions that return indications of failure call **SetLastError** when they fail. They generally do not call **SetLastError** when they succeed. **TlsGetValue** is an exception to this general rule; it calls **SetLastError** to clear a thread's last error when it succeeds. That allows checking for the error-free retrieval of NULL values.

See Also

GetLastError
SetLastError
TlsAlloc
TlsFree
TlsSetValue

TlsSetValue

TlsSetValue stores a value in the calling thread's thread local storage (TLS) slot for a specified TLS index. Each thread of a process has its own slot for each TLS index.

BOOL
TlsSetValue(
DWORD *TlsIndex*,
LPVOID *lpTlsValue*
);

Parameters

TlsIndex

The TLS index that was allocated by **TlsAlloc**.

lpTlsValue

The value to be stored in the calling thread's TLS slot-specified by **TlsIndex**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

TLS indexes are typically allocated by **TlsAlloc** during process or DLL initialization. Once allocated, each thread of the process can use a TLS index to access its own TLS storage slot for that index. The storage slot for each thread is initialized to NULL. A thread specifies a TLS index in a call to **TlsSetValue**, to store a value in its slot. The thread specifies the same index in a subsequent call to **TlsGetValue**, to retrieve the stored value.

TlsSetValue and **TlsGetValue** were implemented with speed as the primary goal. These functions perform minimal parameter validation and error checking. In particular, this function succeeds if TlsIndex is in the range 0 through (TLS_MINIMUM_AVAILABLE - 1). It is up to the programmer to ensure that the index is valid.

See Also

TlsAlloc
TlsFree
TlsGetValue

UnhandledExceptionFilter

UnhandledExceptionFilter displays an Application Error message box and causes the exception handler to be executed. This function can be called only from within the filter expression of a try-except exception handler.

LONG
UnhandledExceptionFilter(
STRUCT_EXCEPTION_POINTERS * *ExceptionInfo*
);

Parameters

ExceptionInfo

A pointer to an EXCEPTION_POINTERS structure containing a description of the exception and the processor context at the time of the exception. This pointer is the return value of a call to the **GetExceptionInformation** function.

Return Values

The function returns one of the following values:

EXCEPTION_CONTINUE_SEARCH
Control returns to the default system exception handler, which terminates the process.

EXCEPTION_EXECUTE_HANDLER
Control returns to the exception handler, which is free to take any appropriate action.

Comments

The function displays an Application Error message box. When a thread of a multi-threaded RTSS process causes an exception, RTSS freezes (or unloads, if so configured) all threads of a process and produces an Application Error message box. The default behavior of Win32 is somewhat different. In Win32, until the user responds to the box, other threads of the Win32 process continue running. When the user has responded, all threads of the Win32 process terminate.

See Also

GetExceptionInformation
SetUnhandledExceptionFilter

WriteFile

WriteFile writes data to a file (synchronous operations only). The function starts writing data to the file at the position indicated by the file pointer. After the write operation has been completed, the file pointer is adjusted by the number of bytes actually written.

```
BOOL
WriteFile(
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped
    );
```

Parameters

hFile

The file to be written to. The file handle must have been created with GENERIC_WRITE access to the file.

lpBuffer

A pointer to the buffer containing the data to be written to the file.

nNumberOfBytesToWrite

The number of bytes to write to the file.

Windows NT and Windows 2000 interpret a value of zero as specifying a null write operation. A null write operation does not write any bytes but does cause the time stamp to change.

lpNumberOfBytesWritten

A pointer to the number of bytes written by this function. **WriteFile** sets this value to zero before doing any work or error checking.

If lpOverlapped is NULL, lpNumberOfBytesWritten cannot be NULL.

lpOverlapped (not supported by RTX)

This parameter must be set to NULL.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

RTX does not support asynchronous operations.

If part of the file is locked by another process and the write operation overlaps the locked portion, this function fails.

Applications must not read from nor write to the output buffer that a write operation is using until the write operation completes. Premature access of the output buffer may lead to corruption of the data written from that buffer.

Windows NT and Windows 2000 interpret zero bytes to write as specifying a null write operation and **WriteFile** does not truncate or extend the file.

See Also

CreateFile
ReadFile

CHAPTER 4

C Run-Time API

Alphabetical List of C Run-Time APIs

The following C run-time library calls are supported in the RTSS environment.

abs	isalpha	memset	strtoul
acos	iscntrl	modf	tan
asin	isdigit	perror	tanh
atan	isgraph	pow	tolower
atan2	islower	printf	toupper
atof	isprint	putc	towlower
atoi	ispunct	putchar	towupper
atol	isspace	qsort	ungetc
bsearch	isupper	rand	va_start
calloc	iswalnum	realloc	vsprintf
ceil	iswalpha	rewind	wcscat
cos	iswascii	setjmp	wcschr
cosh	iswcntrl	signal	wcscmp
difftime	iswctype	sin	wcscpy
div	iswdigit	sinh	wcscspn
exit	iswgraph	sqrt	wcsftime
exp	iswlower	srand	wcslen
fabs	iswprint	sscanf	wcsncat
fclose	iswpunct	strcat	wcsncmp
fflush	iswspace	strchr	wcsncpy
fgets	iswupper	strcmp	wcspbrk
floor	iswxdigit	strcpy	wcsrchr
fmod	isxdigit	strcspn	wcsspn
fopen	labs	strerror	wcsstr
fprintf (stderr)	ldexp	strlen	wcstod
fputc	ldiv	strncat	wcstok
fputs	log	strncmp	wcstol
fread	log10	strncpy	wcstoul
free	longjmp	strpbrk	wmain
frexp	main	strrchr	wprintf
fseek	malloc	strspn	wtof
ftell	memchr	strstr	wtoi
fwrite	memcmp	strtod	wtol
getc	memcpy	strtok	_controlfp
isalnum	memmove	strtol	_fpreset

CHAPTER 5

Windows NT Driver IPC API (RTKAPI) Reference

RtkCloseHandle

RtkCloseHandle closes an open object handle.

BOOL RTKAPI
RtkCloseHandle(
RTSSINST *RtssInst*
PULONG *pErrorCode*
HANDLE *hObject*
);

Parameters

RtssInst

An RTSSINST type returned from call to **RtkRtssAttach**.

pErrorCode

A pointer to Ulong for error returned code.

hObject

An open object handle.

Return Values

If the function succeeds, the return value is TRUE and ErrorCode if defined is set to NULL.

If the function fails, the return value is FALSE. To get any extended error information, check the ErrorCode value. *pErrorCode* may be set to NULL on entry and ignored.

Comments

RtkCloseHandle closes handles to the following RTSS objects:

- Mutex
- Semaphore
- Shared memory
- Event

RtkCloseHandle invalidates the specified object handle, decrements the object's handle count, and performs object retention checks. Once the last handle to an object is closed, the object is removed from the operating system.

RtkCreateEvent

RtkCreateEvent creates an RTSS event. A handle is returned to the newly created event.

HANDLE RTKAPI
RtkCreateEvent(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
LPSECURITY_ATTRIBUTES *pSecurity*,
BOOL *bManualReset*,
BOOL *InitialState*,
PUNICODE_STRING *lpName*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

pSecurity (ignored)

A pointer to a SECURITY_ATTRIBUTES structure.

bManualReset, bInitialState, lpName

A pointer to a PUNICODE_STRING specifying the name of the event object. The name is limited to RTX_MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

Return Values

If the function succeeds, the return value is a handle to the event object. If the named event object existed before the function call, *pErrorCode* is set to ERROR_ALREADY_EXISTS.

If the function fails, the return value is NULL.

Comments

The handle returned by **RtkCreateEvent** has EVENT_ALL_ACCESS access to the new event object and can be used in any function that requires a handle to a event object.

Any thread of the calling process can specify the event-object handle in a call to **RtkWaitForSingleObject**. This wait function returns when the state of the specified object is signaled.

Multiple processes can have handles of the same event object, enabling use of the object for process synchronization. The available object-sharing mechanism is: A process can specify

the name of a event object in a call to **RtkOpenEvent** or **RtkCreateEvent**.

RtkCloseHandle closes an event-object handle. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

See Also

RtkCloseHandle
RtkOpenEvent

RtkCreateMutex

RtkCreateMutex creates an RTSS mutex. A handle is returned to the newly created mutex object.

HANDLE RTKAPI
RtkCreateMutex(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
LPSECURITY_ATTRIBUTES *pSecurity*,
BOOL bInitialOwner,
PUNICODE_STRING *lpName*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

pSecurity (ignored)

A pointer to a SECURITY_ATTRIBUTES structure.

bInitialOwner

The initial ownership state of the mutex object. If TRUE, the calling thread requests immediate ownership of the mutex object. Otherwise, the mutex is not owned.

lpName

A pointer to a PUNICODE_STRING specifying the name of the mutex object. The name is limited to RTX_MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

If *lpName* matches the name of an existing named mutex object, this function requests MUTEX_ALL_ACCESS access to the existing object. In this case, the *bInitialOwner* parameter is ignored because it has already been set by the creating process.

If *lpName* matches the name of an existing semaphore, the function fails; ERROR_INVALID_HANDLE is returned in the ErrorCode location, if defined, provided by the caller. This occurs because mutex and semaphore objects share the same name space.

Return Values

If the function succeeds, the return value is a handle to the mutex object. If the named mutex object existed before the function call, *pErrorCode* is set to ERROR_ALREADY_EXISTS.

If the function fails, the return value is NULL.

Comments

The handle returned by **RtkCreateMutex** has MUTEX_ALL_ACCESS access to the new mutex object and can be used in any function that requires a handle to a mutex object.

Any thread of the calling process can specify the mutex-object handle in a call to **RtkWaitForSingleObject**. This wait function returns when the state of the specified object is signaled.

The state of a mutex object is signaled when it is not owned by any thread. The creating thread can use the *bInitialOwner* flag to request immediate ownership of the mutex. Otherwise, a thread must use the wait function to request ownership. When the mutex's state is signaled, the highest priority waiting thread is granted ownership (if more than one thread is waiting at the same priority, they receive ownership of the mutex in the order they waited); the mutex's state changes to non-signaled; and the wait function returns. Only one thread can own a mutex at any given time. The owning thread uses **RtkReleaseMutex** to release its ownership.

The thread that owns a mutex can specify the same mutex in repeated wait function calls without blocking its execution. Typically, you would not wait repeatedly for the same mutex, but this mechanism prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call **RtkReleaseMutex** once for each time that the mutex satisfied a wait.

Two or more processes can call **RtkCreateMutex** to create the same named mutex. The first process actually creates the mutex, and subsequent processes open a handle to the existing mutex. This enables multiple processes to get handles of the same mutex, while relieving the user of the responsibility of ensuring that the creating process is started first. When using this technique, you should set the *bInitialOwner* flag to FALSE; otherwise, it can be difficult to be certain which process has initial ownership.

Multiple processes can have handles of the same mutex object, enabling use of the object for process synchronization. The available object-sharing mechanism is: A process can specify the name of a mutex object in a call to **RtkOpenMutex** or **RtkCreateMutex**.

RtkCloseHandle closes a mutex-object handle. The system closes the handle automatically when the process terminates. The mutex object is destroyed when its last handle has been closed.

See Also

RtkCloseHandle
RtkOpenMutex
RtkReleaseMutex

RtkCreateSemaphore

RtkCreateSemaphore creates a named semaphore object.

HANDLE RTKAPI
RtkCreateSemaphore(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
LPSECURITY_ATTRIBUTES *pSecurity*,
LONG *lInitialCount*,
LONG *lMaximumCount*,
PUNICODE_STRING *lpName*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

pSecurity (ignored)

A pointer to a SECURITY_ATTRIBUTES structure.

lpSemaphoreAttributes (ignored)

A pointer to security attributes.

lInitialCount

An initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to *lMaximumCount*. The state of a semaphore is signaled when its count is greater than zero and non-signaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling **RtkReleaseSemaphore**.

lMaximumCount

The maximum count for the semaphore object. This value must be greater than zero.

lpName

A pointer to a PUNICODE_STRING specifying the name of the mutex object. The name is limited to RTX_MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

If *lpName* matches the name of an existing named semaphore object, this function requests access to the existing object. In this case, *lInitialCount* and *lMaximumCount* are ignored

because they have already been set by the creating process.

Return Values

If the function succeeds, the return value is a handle to the semaphore object. If *lpName* matches the name of an existing semaphore, the function fails; ERROR_INVALID_HANDLE is returned in the ErrorCode location, if defined, provided by the caller. This occurs because mutex and semaphore objects share the same name space.

Comments

The handle returned by **RtkCreateSemaphore** has all accesses to the new semaphore object and can be used in any function that requires a handle to a semaphore object.

Any thread of the calling process can specify the semaphore-object handle in a call to one of the wait functions. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution.

The state of a semaphore object is signaled when its count is greater than zero, and non-signaled when its count is equal to zero. *lInitialCount* specifies the initial count. Each time a waiting thread is released because of the semaphore's signaled state, the count of the semaphore is decreased by one. Use **RtkReleaseSemaphore** to increment a semaphore's count by a specified amount. The count can never be less than zero or greater than the value specified in *lMaximumCount*.

Multiple processes can have handles of the same semaphore object, enabling use of the object for inter-process synchronization. The available object-sharing mechanism is: A process can specify the name of a semaphore object in a call to **RtkOpenSemaphore** or **RtkCreateSemaphore**.

Use **RtkCloseHandle** to close the handle. The system closes the handle automatically when the process terminates. The semaphore object is destroyed when its last handle has been closed.

See Also

RtkCloseHandle
RtkOpenSemaphore
RtkReleaseSemaphore

RtkCreateSharedMemory

RtkCreateSharedMemory creates a named region of physical memory that can be mapped by any process.

HANDLE RTKAPI
RtkCreateSharedMemory(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
DWORD *flProtect*,
DWORD *MaximumSizeHigh*,
DWORD *MaximumSizeLow*,
PUNICODE_STRING *lpName*,
VOID ** *location*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

flProtect (ignored by RTSS)

The protection desired for the shared memory view. This parameter can be one of the following values:

PAGE_READONLY
Gives read-only access to the committed region of pages. An attempt to write to or execute the committed region results in an access violation.

PAGE_READWRITE
Gives read-write access to the committed region of pages.

MaximumSizeHigh

The high-order 32 bits of the size of the shared memory object.

MaximumSizeLow

The low-order 32 bits of the size of the shared memory object.

lpName

A pointer to a PUNICODE_STRING specifying the name of the shared memory object. The name is limited to RTX_MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

If this parameter matches the name of an existing named shared memory object, the function

requests access to the shared memory object with the protection specified by *flProtect*.

location

A pointer to a location where the virtual address of the shared memory will be stored.

Return Value

If the function succeeds, the return value is a handle to the shared memory object. If the object existed before the function call, ErrorCode, if defined, contains ERROR_ALREADY_EXISTS, and the return value is a valid handle to the existing shared memory object (with its current size, not the new specified size).

If the function fails, the return value is NULL.

Comments

The handle that **RtkCreateSharedMemory** returns has full access to the new shared memory object. Shared memory objects can be shared by name. For information on opening a shared memory object by name, see **RtkOpenSharedMemory**.

To fully close a shared memory object, an application must close the physical mapping object handle by calling **RtkCloseHandle**. The order in which these functions are called does not matter.

When all handles to the shared memory object representing the physical memory are closed, the object is destroyed and physical memory is returned to the system.

See Also

RtkCloseHandle
RtkOpenSharedMemory

RtkOpenEvent

RtkOpenEvent returns a handle to the named RTSS event.

HANDLE RTKAPI
RtOpenEvent(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
DWORD *DesiredAccess*,
BOOL *bInheritHandle*,
PUNICODE_STRING *lpName*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

DesiredAccess (ignored)

bInheritHandle (ignored)

lpName

A pointer to a PUNICODE_STRING specifying the name of the event object. The name is limited to RTX_MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

Return Values

If the function succeeds, the return value is a handle of the event object.

If the function fails, the return value is NULL.

Comments

RtkOpenEvent enables multiple processes to open handles of the same event object. The function succeeds only if some process has already created the event with **RtkCreateEvent**. The calling process can use the returned handle in any function that requires a handle of a event object, such as a wait function.

Use **RtkCloseHandle** to close the handle. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

See Also

RtkCloseHandle
RtkCreateEvent

RtkOpenMutex

RtkOpenMutex returns a handle to the named RTSS mutex.

HANDLE

RtOpenMutex(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
DWORD *DesiredAccess*,
BOOL *bInheritHandle*,
PUNICODE_STRING *lpName*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

DesiredAccess (ignored)

The requested access to the mutex object.

bInheritHandle (ignored)

An indicator whether the returned handle is inheritable.

lpName

A pointer to a PUNICODE_STRING specifying the name of the mutex object. The name is limited to RTX_MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

Return Values

If the function succeeds, the return value is a handle of the mutex object.

If the function fails, the return value is NULL.

Comments

RtkOpenMutex enables multiple processes to open handles of the same mutex object. The function succeeds only if some process has already created the mutex with **RtkCreateMutex**. The calling process can use the returned handle in any function that requires a handle of a mutex object, such as a wait function.

Use **RtkCloseHandle** to close the handle. The system closes the handle automatically when the process terminates. The mutex object is destroyed when its last handle has been closed.

See Also

RtkCloseHandle
RtkCreateMutex
RtkReleaseMutex

RtkOpenSemaphore

RtkOpenSemaphore returns a handle of an existing named semaphore object.

HANDLE RTKAPI
RtkOpenSemaphore(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
DWORD *DesiredAccess*,
BOOL *bInheritHandle*,
PUNICODE_STRING *lpName*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

DesiredAccess

The requested access to the semaphore object. This parameter can be any combination of the following values:

SEMAPHORE_ALL_ACCESS
Specifies all possible access flags for the semaphore object.

SEMAPHORE_MODIFY_STATE
Enables use of the semaphore handle in **RtkReleaseSemaphore** to modify the semaphore's count.

SYNCHRONIZE
Enables use of the semaphore handle in any of the wait functions to wait for the semaphore's state to be signaled.

bInheritHandle

This must be FALSE.

lpName

A pointer to a PUNICODE_STRING specifying the name of the mutex object. The name is limited to RTX_MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

Return Values

If the function succeeds, the return value is a handle of the semaphore object.

If the function fails, the return value is NULL.

Comments

RtkOpenSemaphore enables multiple processes to open handles of the same semaphore object. The function succeeds only if some process has already created the semaphore by using **RtkCreateSemaphore**. The calling process can use the returned handle in any function that requires a handle of a semaphore object, such as a wait function, subject to the limitations of the access specified in *DesiredAccess*.

Use **RtkCloseHandle** to close the handle. The system closes the handle automatically when the process terminates. The semaphore object is destroyed when its last handle has been closed.

See Also

RtkCloseHandle
RtkReleaseSemaphore

RtkOpenSharedMemory

RtkOpenSharedMemory opens a named physical-mapping object.

HANDLE RTKAPI
RtkOpenSharedMemory(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
DWORD *DesiredAccess*,
BOOL *bInheritHandle*,
PUNICODE_STRING *lpName*,
VOID ** *location*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

DesiredAccess

The access mode. The RTSS environment always grants read and write access. This parameter can be one of the following values:

SHM_MAP_WRITE
Read-write access. The target shared memory object must have been created with PAGE_READWRITE protection. A read-write view of the shared memory is mapped.

SHM_MAP_READ
Read-only access. The target shared memory object must have been created with PAGE_READWRITE or PAGE_READ protection. A read-only view of the shared memory is mapped.

bInheritHandle (ignored)

lpName

A pointer to a PUNICODE_STRING specifying the name of the shared memory object. The name is limited to RTX_MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case-sensitive.

location

A pointer to a location where the virtual address of the mapping will be stored.

Return Values

If the function succeeds, the return value is an open handle to the specified shared memory object.

If the function fails, the return value is NULL.

Comments

The handle that **RtkOpenSharedMemory** returns can be used with **RtkCloseHandle** to decrement the reference count to the shared memory object. When the reference count is zero, the object is removed from the system.

See Also

RtkCreateSharedMemory
RtkCloseHandle

RtkPulseEvent

RtkPulseEvent provides a single operation that sets (to signaled) the state of the specified event object and then resets it (to non-signaled) after releasing the appropriate number of waiting threads.

BOOL RTKAPI
RtkPulseEvent(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
HANDLE *hEvent*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

hEvent

The handle which identifies the event object as returned by a preceding call to **RtkCreateEvent** or **RtkOpenEvent**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Comments

For a manual-reset event object, all waiting threads that can be released are released. The function then resets the event object's state to non-signaled and returns.

For an auto-reset event object, the function resets the state to non-signaled and returns after releasing a single waiting thread, even if multiple threads are waiting.

If no threads are waiting, or if no thread can be released immediately, **RtkPulseEvent** simply sets the event object's state to non-signaled and returns.

See Also

RtkCreateEvent
RtkOpenEvent
RtkWaitForSingleObject

RtkReleaseMutex

RtkReleaseMutex relinquishes ownership of an RTSS mutex.

BOOL RTKAPI
RtkReleaseMutex(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
HANDLE *hMutex*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

hMutex

The handle which identifies the mutex object as returned by a preceding call to **RtkCreateMutex** or **RtkOpenMutex**.

Return Values

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

RtkReleaseMutex fails if the calling thread does not own the mutex object.

A thread gets ownership of a mutex by specifying a handle of the mutex in **RtkWaitForSingleObject**. The thread that creates a mutex object can also get immediate ownership without using one of the wait functions. When the owning thread no longer needs to own the mutex object, it calls **RtkReleaseMutex**.

While a thread has ownership of a mutex, it can specify the same mutex in additional wait-function calls without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call **RtkReleaseMutex** once for each time that the mutex satisfied a wait.

See Also

RtkCreateMutex
RtkOpenMutex
RtkWaitForSingleObject

RtkReleaseSemaphore

RtkReleaseSemaphore increases the count of the specified semaphore object by a specified amount.

BOOL RTKAPI
RtkReleaseSemaphore(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
HANDLE *hSemaphore*,
LONG *lReleaseCount*,
PLONG *lpPreviousCount*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

hSemaphore

The semaphore object. **RtkCreateSemaphore** or **RtkOpenSemaphore** returns this handle.

lReleaseCount

The amount by which the semaphore object's current count is to be increased. The value must be greater than zero. If the specified amount would cause the semaphore's count to exceed the maximum count that was specified when the semaphore was created, the count is not changed and the function returns FALSE.

lpPreviousCount

A pointer to a 32-bit variable receives the previous count for the semaphore. This parameter can be NULL if the previous count is not required.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Comments

The state of a semaphore object is signaled when its count is greater than zero and non-signaled when its count is equal to zero. The process that calls **RtkCreateSemaphore** specifies the semaphore's initial count. Each time a waiting thread is released because of the

semaphore's signaled state, the count of the semaphore is decreased by one.

Typically, an application uses a semaphore to limit the number of threads using a resource. Before a thread uses the resource, it specifies the semaphore handle in a call to one of the wait functions. When the wait function returns, it decreases the semaphore's count by one. When the thread has finished using the resource, it calls **RtkReleaseSemaphore** to increase the semaphore's count by one.

Another use of **RtkReleaseSemaphore** is during an application's initialization. The application can create a semaphore with an initial count of zero. This sets the semaphore's state to non-signaled and blocks all threads from accessing the protected resource. When the application finishes its initialization, it uses **RtkReleaseSemaphore** to increase the count to its maximum value, to permit normal access to the protected resource.

See Also

RtkCreateSemaphore
RtkOpenSemaphore

RtkResetEvent

RtkResetEvent provides a single operation that sets (to signaled) the state of the specified event object and then resets it (to non-signaled) after releasing the appropriate number of waiting threads.

BOOL RTKAPI
RtkResetEvent(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
HANDLE *hEvent*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

hEvent

The handle which identifies the event object as returned by a preceding call to **RtkCreateEvent** or **RtkOpenEvent**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Comments

For a manual-reset event object, all waiting threads that can be released are released. The function then resets the event object's state to non-signaled and returns.

For an auto-reset event object, the function resets the state to non-signaled and returns after releasing a single waiting thread, even if multiple threads are waiting.

If no threads are waiting, or if no thread can be released immediately, **RtkPulseEvent** simply sets the event object's state to non-signaled and returns.

See Also

RtkCreateEvent
RtkOpenEvent
RtkWaitForSingleObject

RtkRtssAttach

RtkRtssAttach attaches a kernel device driver to RTSS.

RTSSINST RTKAPI
RtkRtssAttach(
LONG *MaxWFSO*,
PULONG *pErrorCode*,
);

Parameters

MaxWFSO

The count of the number of wait-for-single-objects the user requires. The required minimum is one.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

Return Values

If the function succeeds, the return value is a non-zero RTSSINST.

If the function fails, the return value is zero.

Comments

This call is made only once by a kernel-resident device driver. The returned instance must be used for all subsequent RTKAPI calls. This call is usually made at, but not restricted to, driver entry.

If the device driver caller starts at boot time - you have to, first, set RTX to start at the boot time, and second, make this call ONCE in the Driver Dispatch routine, NEVER in the DriverEntry() routine.

See Also

RtkRtssDetach

RtkRtssDetach

RtkRtssDetach detaches a kernel device driver from RTSS.

VOIDRTKAPI
RtkRtssAttach(
RTSSINST *RtssInst*
PULONG *pErrorCode*,
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

Return Values

The function always succeeds.

Comments

This call is made only once by a kernel-resident device driver to disconnect or detach from RTSS. This call is usually made at, but not restricted to, driver unload.

See Also

RtkRtssAttach

RtkSetEvent

RtkSetEvent sets the state of the specified event object to signaled.

BOOL RTKAPI
RtkSetEvent(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
HANDLE *hEvent*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

hEvent

The handle which identifies the event object as returned by a preceding call to **RtkCreateEvent** or **RtkOpenEvent**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Comments

The state of a manual-reset event object remains signaled until it is set explicitly to the non-signaled state by the **RtkSetEvent** function. Any number of waiting threads, or threads that subsequently begin wait operations for the specified event object by calling the wait functions, can be released while the object's state is signaled.

The state of an auto-reset event object, the function resets the state to non-signaled and returns after releasing remains signaled until a single waiting thread is released, at which time the system automatically sets the state to non-signaled. If no threads are waiting, the event object's state remains signaled.

See Also

RtkCreateEvent
RtkOpenEvent
RtkWaitForSingleObject

RtkWaitForSingleObject

RtkWaitForSingleObject returns when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses.

ULONG
RtkWaitForSingleObject(
RTSSINST *RtssInst*,
PULONG *pErrorCode*,
HANDLE *hHandle*,
DWORD *Milliseconds*
);

Parameters

RtssInst

An RTSSINST value returned from a call to **RtkRtssAttach**.

pErrorCode

A pointer to a location where additional error information may be returned. This location need not be defined; the user may pass a NULL value. If defined, this location is set to NULL if no error occurred.

hHandle

The object identifier. See the list of the object types whose handles can be specified in the Comments section.

Milliseconds

The time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is non-signaled. If *Milliseconds* is zero, the function tests the object's state and returns immediately. If *Milliseconds* is INFINITE, the function's time-out interval never elapses.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return.

If the function fails, the return value is WAIT_FAILED.

The return value on success is one of the following values:

WAIT_ABANDONED
The specified object is a mutex object that was not released by the thread that owned the mutex object before the owning thread terminated. Ownership of the mutex object is granted to the calling thread, and the mutex is set to non-signaled.

WAIT_OBJECT_0
The state of the specified object is signaled.

WAIT_TIMEOUT
The time-out interval elapsed, and the object's state is non-signaled.

Comments

RtkWaitForSingleObject checks the current state of the specified object. If the object's state is non-signaled, the calling thread enters an efficient wait state. The thread consumes very little processor time while waiting for the object state to become signaled or the time-out interval to elapse.

Before returning, a wait function modifies the state of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. For example, the count of a semaphore object is decreased by one.

RtkWaitForSingleObject can wait for the following objects:

Semaphore
RtkCreateSemaphore or **RtkOpenSemaphore** returns the handle. A semaphore object maintains a count between zero and some maximum value. Its state is signaled when its count is greater than zero and non-signaled when its count is zero. If the current state is signaled, the wait function decreases the count by one.

Mutex
RtkCreateMutex and **RtkOpenMutex** return handles to the mutex object which becomes signaled when the mutex is unowned.

See Also

RtkCreateMutex
RtkCreateSemaphore
RtkOpenMutex
RtkOpenSemaphor

Index

S

T

U

V

W