

About INtime Win32 and real time extension system calls

Contents

- Overview
- How the Win32 API (iWin32) fits in the INtime environment
- INtime Win32 system calls
- Porting a Windows application to INtime

Overview

The INtime API was originally designed to access INtime functions from a process running in the isolated INtime environment and is intentionally similar but not identical to the Win32 API used in 32-bit Windows programming.

For developers familiar with the Windows environment, the 'iWin32' (INtime subset of Win32) API is now available, an alternative that includes a subset of the Win32 functions, with these main elements:

- Processes and threads
- Mutexes, critical sections, semaphores and events
- Structured exception handling (SEH)
- I/O handling

In addition a number of real-time extension (RTX) functions are provided where more real-time functionality is required; this includes functions for:

- Interrupt handling
- Shared memory
- Timers

In many applications it is necessary to communicate between an INtime process and a Windows process. When using the INtime API, NTX (an extension to the Windows API) is provided, which is culturally compatible with the INtime API.

If the INtime application uses the iwin32 API, a corresponding set of functions (named iwin32x) is available for use in the Windows environment; the names of these functions consist of the name of the equivalent iwin32 function with a Rt prefix.

How the Win32 API (iwin32) fits in the INtime environment

iWin32 provides an alternative to the standard INtime API. Iwin32 includes a number of objects that are similar to INtime objects and some that are specific to iwin32; INtime also provides some objects for which iwin32 provides no interface and in that case API mixing may be used.

We now get to the situation that there can be more than one API choice for a specific function (for example: CreateSemaphore, CreateRtSemaphore or RtCreateSemaphore). All APIs can be mixed with the restriction that once a handle to an object is obtained with one particular API choice, that API must be applied to all uses of the particular handle. For example, if the handle is obtained with CreateSemaphore, you can use ReleaseSemaphore, WaitForSingleObject and CloseHandle but not ReleaseRtSemaphore, WaitForRtSemaphore and DeleteRtSemaphore. Further restrictions are explained in the paragraph on API mixing.

Here is an overview of the major elements of the iwin32 API:

- [Handle](#)
- [Named Object](#)
- [Process](#)
- [Thread](#)
- [Mutex](#)
- [Critical section](#)
- [Semaphore](#)
- [Event](#)
- [I/O handling](#)
- [Interrupt handling](#)
- [Shared memory](#)
- [Timer](#)
- [Exception handling](#)

[Miscellaneous](#)
[Exception codes](#)
[Project settings](#)
[API mixing](#)
[Iwin32x functions](#)

Handles

Each object is identified by a handle. In INtime an object is uniquely identified by a single handle value (16 bits for INtime, 32 bits for NTX). This handle can be used in any INtime process and in Win32 processes (using NTX). When the object is deleted with a type-specific deletion function such as [DeleteRtSemaphore](#), the handle becomes invalid.

iWin32 has a different handle system: the Create and Open functions return a handle and different callers may receive different handles for the same object. A handle is stored in 32 bits; an iwin32 handle can be distinguished from an INtime handle because its value is 64K or greater. Every iwin32 object includes a handle count. When the last handle for an object is closed, the object is implicitly deleted.

In Win32, a handle is normally specific to a process and the same handle in different Win32 processes may refer to different objects. Iwin32 implements a slightly different method, where all handle values are unique. This allows a handle to be shared between processes, which would be against the Win32 rules.

There is a limit on the number of objects that can exist at any time in the system, because INtime uses a table to define each object; the size of this table ([GDT](#)) is configurable up to a maximum of 8000 entries. Each iwin32 object requires one, two (thread, timer, interrupt) or three (process) INtime objects. Additional handles for a given iwin32 object do not require additional INtime objects. Iwin32 uses a fixed size table for all handles, the size of which is configurable.

Functions that operate on handles are:

[CloseHandle](#) and [RtCloseHandle](#)

Named object

Event, mutex, semaphore and shared memory objects have a Create and an Open function. CreateXxx checks if the named object of that type already exists and if so, returns an error *and* the handle of the found object. If the name exists but belongs to another object type, the function fails. If the name does not occur yet, the object is created and the name remembered. If no name is supplied, the name check does not take place. OpenXxx only does the name check and if that fails, the whole operation fails.

All object types share one name space, which is not process specific but has system scope. Iwin32 allows names up to 128 characters.

There are no specific functions for named objects, so for details look at the different object types.

Process

A process is a container for objects and resources; it includes a virtual address space that is only accessible to the threads in the process. When a process is created, a primary thread is always created inside it (this is the function named **main**).

A process can refer to itself by a so-called pseudo handle, which is not a fixed value, but must be obtained by the [GetCurrentProcess](#) function. A pseudo handle can only be used by the process itself, and it cannot be closed (it is implicitly closed when the process terminates).

A process can be explicitly terminated with the [TerminateProcess](#) or [ExitProcess](#) functions; implicit termination obeys these rules:

- 1 When the primary thread returns, the process is terminated
- 2 When the primary thread calls [ExitThread](#) explicitly, the process is not terminated.
- 3 When any thread terminates and it was the last thread in the process, the process is terminated.

Terminating a process does not necessarily delete the process! It only closes the pseudo handle for the process and only if that is the last handle, the process is deleted. When a process is terminated, all handles created by its threads are closed; again, this need not imply that all *objects* are deleted.

Waiting for a process to be signaled means waiting until the process has terminated.

Process functions include:

[ExitProcess](#)
[GetCurrentProcess](#)
[GetCurrentProcessId](#)

[GetExitCodeProcess](#) and [RtGetExitCodeProcess](#)
[OpenProcess](#) and [RtOpenProcess](#)
[TerminateProcess](#) and [RtTerminateProcess](#)
[RtCreateProcess](#) (in iwin32x)
[WaitforMultipleObjects](#) and [RtWaitforMultipleObjects](#)
[WaitForSingleObject](#) and [RtWaitForSingleObject](#)

Thread

A thread is the active element type in the system. Each thread has a priority, a state and a stack. The priority indicates the importance of the thread when it is in the ready state.

A thread is in one of these states:

ready	the thread wants to execute; out of the set of ready threads (called the ready list) the thread with the best priority becomes the running thread.
asleep	the thread waits for an object or one of a set of objects to be signaled, or for a specific timeout, or both. While in this state, the thread will never be running.
suspended	the thread is waiting for a resume operation. More than one suspend can be done, and each such suspend must be undone by a resume. While in this state, the thread will never be running.
asleep suspended	while in the asleep state, the thread was suspended. Both the suspend state and the asleep state must be undone before the thread becomes ready again.

A thread has a stack for calling functions and storing local variables and parameters. The stack must be big enough to contain all necessary data; when it overflows, it is not extended but the hardware exception EH_STACK_FAULT occurs.

A thread can refer to itself by a so-called pseudo handle, which is not a fixed value, but must be obtained by the [GetCurrentThread](#) function. A pseudo handle can only be used within the owning process, and it cannot be closed (it is implicitly closed when the thread terminates).

Waiting for a thread to be signaled means waiting until the thread has terminated.

Thread handling functions include:

[CreateThread](#)
[ExitThread](#)
[GetCurrentThread](#)
[GetCurrentThreadId](#)
[GetExitCodeThread](#)
[GetLastError](#)
[GetThreadPriority](#) and [RTGetThreadPriority](#)
[RtGetThreadTimeQuantum](#)
[OpenThread](#)
[ResumeThread](#)
[SetLastError](#)
[SetThreadPriority](#) and [RtSetThreadPriority](#)
[RtSetThreadTimeQuantum](#)
[Sleep](#)
[RtSleepFt](#)
[SuspendThread](#)
[TerminateThread](#)
[WaitForMultipleObjects](#) and [RtWaitForMultipleObjects](#)
[WaitForSingleObject](#) and [RtWaitForSingleObject](#)

Mutex

A mutex is an object for getting exclusive access to a resource used by more than one thread, possibly in different processes.

When a thread attempts to get ownership of a mutex and that mutex is free, ownership is given to the owner; until the thread releases ownership, no other thread can own the same mutex. A thread can own the same mutex more than once, in which case it must release the mutex the same number of times.

When a thread with INtime [priority](#) Pw wishes to own a mutex and that mutex is already owned by another thread with priority Po, then if Pw < Po, the priority of the owning thread is changed to Pw until it releases all mutexes it owns. This avoids the infamous [priority inversion](#).

Termination of a thread that owns one or more mutexes causes all threads waiting for such mutexes to be woken up with a WAIT_ABANDONED exception. Deleting a mutex causes all threads waiting for that mutex to be woken up with an ERROR_INVALID_HANDLE error code.

Mutex manipulation functions are:

[CreateMutex](#) and [RtCreateMutex](#)
[OpenMutex](#) and [RtOpenMutex](#)
[ReleaseMutex](#) and [RtReleaseMutex](#)
[WaitforMultipleObjects](#) and [RtWaitforMultipleObjects](#)
[WaitForSingleObject](#) and [RtWaitForSingleObject](#)

Critical section

A critical section is a mutex that has no name; it can therefore only be used in the process that creates it. A critical section is identified by a CRITICAL_SECTION structure, which in turn contains the handle of the mutex. For more details see [mutexes](#).

Critical sections are not available in the iwin32x API (but since mutexes are, they can be easily simulated).

Functions for critical sections include:

[DeleteCriticalSection](#)
[EnterCriticalSection](#)
[InitializeCriticalSection](#)
[LeaveCriticalSection](#)
[TryEnterCriticalSection](#)

Semaphore

A semaphore is a counter that takes positive integer values called units. Threads release units to and wait for units from the semaphore. A semaphore can synchronize a thread's actions with other threads and can also be used to provide mutual exclusion for data or a resource (although a mutex may be better in that case).

A thread can release one or more units to a semaphore. Waiting can be done for a single unit only. A semaphore does not protect against [priority inversion](#).

Deleting a semaphore causes all threads waiting for that semaphore to be woken up with an ERROR_INVALID_HANDLE error code.

Semaphore functions include:

[CreateSemaphore](#) CreateSemaphore>main and [RtCreateSemaphore](#) CreateSemaphore>main
[OpenSemaphore](#) OpenSemaphore>main and [RtOpenSemaphore](#) OpenSemaphore>main
[ReleaseSemaphore](#) ReleaseSemaphore>main and [RtReleaseSemaphore](#) ReleaseSemaphore>main
[WaitforMultipleObjects](#) and [RtWaitforMultipleObjects](#)
[WaitForSingleObject](#) and [RtWaitForSingleObject](#)

Event

An event is a flag that can be set (signaled) or reset; it can be reset manually (once set, it remains set until explicitly reset by a [ResetEvent](#) call, independent of how many threads are woken up) or automatically (after waking up one thread, the event is reset).

Deleting an event causes all threads waiting for that event to be woken up with an ERROR_INVALID_HANDLE error code.

Event functions include:

[CreateEvent](#) and [RtCreateEvent](#)
[OpenEvent](#) and [RtOpenEvent](#)
[PulseEvent](#) and [RtPulseEvent](#)
[SetEvent](#) and [RtSetEvent](#)
[WaitforMultipleObjects](#) and [RtWaitforMultipleObjects](#)
[WaitForSingleObject](#) and [RtWaitForSingleObject](#)

I/O handling

In iWin32 a few general file handling functions are present. For many functions, the C-library offers alternatives. Device dependent functions (as provided by DeviceIoControl in Win32) can either be programmed using port I/O, or can be delegated to INtime device drivers.

In contrast to Win32, port I/O (accessing hardware ports directly) is allowed in all INtime threads.

I/O functions in iwin32 include:

[CreateFile](#)
[DeleteFile](#)
[RtDisablePortIo](#)
[RtEnablePortIo](#)
[RtGetBusDataByOffset](#)
[ReadFile](#)
[RtReadPortXxx](#)
[RemoveDirectory](#)
[RtSetBusDataByOffset](#)
[SetFilePointer](#)
[WriteFile](#)
[RtWritePortXxx](#)

Interrupt handling

Win32 does not provide interrupt handling functions, as this always takes place in the Windows kernel environment. Since interrupts are critical in INtime, we have extended iwin32 with interrupt handling. There are two choices for handling an interrupt:

- **Using [RtAttachInterruptVector](#):** a thread is created that is woken up when an interrupt occurs. The thread may use all INtime functions, which makes this a simple-to-understand approach. There is a penalty in processing time, as each interrupt requires two thread switches for switching to and from the interrupt thread.
- **Using [RtAttachInterruptVectorEx](#):** as with the previous function, a thread is created. But in addition a function may be specified that gets called from the hardware interrupt handler, which then determines the need to wake up the thread. In this way many thread switches can be avoided, such as in the case of a terminal: the interrupt function can cause thread wake up for a carriage return character and do internal buffering (and maybe editing) for all other characters. Such an interrupt function can only use the I/O functions [RtReadPortXxx](#) and [RtWritePortXxx](#).

When an interrupt comes from a PCI source, the actual interrupt line can be determined using [RtGetBusDataByOffset](#). Access to I/O ports on the device for determining interrupt details is provided by the [RtReadPortXxx](#) and [RtWritePortXxx](#) functions.

The thread created for interrupt handling is a special one: it can not be suspended or resumed and its priority can not be changed. It should not call [ExitThread](#) and can not be terminated by [TerminateThread](#).

Interrupt handling functions include:

[RtAttachInterruptVector](#)
[RtAttachInterruptVectorEx](#)
[RtDisableInterrupts](#)
[RtEnableInterrupts](#)
[RtReleaseInterruptVector](#)

Shared memory

Mutexes and semaphores allow threads to synchronize, but what if you want to exchange data? You can use INtime objects such as mailboxes, but in iwin32 you also find shared memory. Shared memory is memory that has been allocated by one process and that can be accessed by other processes as well. To access shared memory created by another process you need to know its name.

Since every process has its own virtual address space, a shared memory object must be mapped into a process' address space. Different processes may use different local addresses to access the same shared memory! The shared memory is only deleted when all its handles are closed.

Space for shared memory objects comes from an iwin32 virtual memory pool; the maximum size of that pool is configurable.

It is up to the communicating threads to agree on a method of queuing data in the shared memory as necessary.

Shared memory functions include:

[RtCreateSharedMemory](#)
[RtOpenSharedMemory](#)

Win32 defines its own exception codes, which can overlap with INtime numbers. Fortunately, both environments

Timer

Any thread can be made to wait for a given time by using [Sleep](#) or [RtSleepFt](#). A timer is simply a thread that gets woken up when its time passes. Creating a timer means that a thread is created that calls a user-provided function after a given time. This thread is a special one: it can not be suspended or resumed and its priority can not be changed. It should not call [ExitThread](#) and can not be terminated by [TerminateThread](#).

Timer functions include:

[RtCancelTimer](#)
[RtCreateTimer](#)
[RtDeleteTimer](#)
[RtGetClockResolution](#)
[RtGetClockTime](#)
[RtGetClockTimerPeriod](#)
[RtGetTimer](#)
[RtSetClockTime](#)
[RtSetTimer](#)
[RtSetTimerRelative](#)

Exception handling

Iwin32 exception handling allows a thread to deal with its own hardware exceptions (exceptions occurring in system calls are not included). There are two types of exception handling:

Structured Exception Handling or SEH: using `__try` and `__except` blocks and some APIs like `GetExceptionCode` and `RaiseException`, the programmer can decide how a thread reacts to exceptions. An (fairly trivial) example follows:

```
int Filter(LPEXCEPTION_POINTERS pEP) {
    printf("Exception %x at address %x\n",
           pEP->ExceptionRecord.ExceptionCode,
           pEP->ExceptionRecord.ExceptionAddress);
    return EXCEPTION_EXECUTE_HANDLER;
}

__try {
    perform some process that may cause a hardware exception
}
__except(Filter(GetExceptionInformation())) {
    ExitProcess(1);
}
```

C++ exception handling: C++ provides `try` and `catch` blocks and `throw` statements to handle exceptions. Although there are syntactical differences from SEH, semantically this is similar to SEH. A similar example could be:

```
try {
    perform some process that may cause a hardware exception
}
catch(EXCEPTION e) {
    printf("Exception %x at address %x\n",
           e.ExceptionCode,
           e.ExceptionAddress);
    ExitProcess(1);
}
```

For details, see the language definition for C/C++ and the Microsoft Visual Studio documentation.

Exception handling functions include:

[_set_se_translator](#)
[AbnormalTermination](#)
[GetExceptionCode](#)
[GetExceptionInformation](#)

Miscellaneous

A number of functions have been added to iwin32 because they are very useful for real-time applications or to provide source compatibility with existing applications.

- Physical memory functions allow mapping of physical memory to virtual addresses, which may be required for access to hardware components:

[RtGetPhysicalAddress](#)
[RtMapMemory](#)
[RtUnmapMemory](#)

- Memory allocation is already available via [malloc](#) and [free](#), but for compatibility the following functions are also provided:

[RtAllocateContiguousMemory](#)
[RtAllocateLockedMemory](#)
[RtFreeContiguousMemory](#)
[RtFreeLockedMemory](#)
[HeapAlloc](#)
[HeapFree](#)
[HeapReAlloc](#)
[HeapSize](#)

- Real-time Shared Library (RSL) functions help in the use of RSLs at execution time (load time use of RSLs does not require any functions). For details on these functions see the equivalent INtime function (insert Rt after the first syllable) or the Win32 function with the same name:

[FreeLibrary](#)
[GetModuleHandle](#)
[GetProcAddress](#)
[LoadLibrary](#)

- Quick synchronization functions allow do-it-yourself synchronization without the expense of a system call:

[InterlockedCompare](#)
[InterlockedCompareExchange](#)
[InterlockedCompareExchangePointer](#)
[InterlockedDecrement](#)
[InterlockedExchange](#)
[InterlockedExchangeAdd](#)
[InterlockedExchangePointer](#)
[InterlockedIncrement](#)

- The following functions perform no action, but are provided for source compatibility with existing applications:

[RtCommitLockHeap](#)
[RtCommitLockProcessHeap](#)
[RtCommitLockStack](#)
[RtDisablePortIo](#)
[RtEnablePortIo](#)
[RtLockKernel](#)
[RtLockProcess](#)
[RtUnlockKernel](#)
[RtUnlockProcess](#)

Exception codes

Win32 defines its own exception codes, which can overlap with INtime numbers. Fortunately, both environments define zero as 'no error'. The INtime kernel stores only 16 bits which is sufficient for the basic Win32 exception codes (only COM and some other Win32 subsystems need more bits).

Hardware exceptions have different exception codes in the two environments. The following translation is applied to hardware exceptions (this is only relevant for [Structured Exception Handling and C++ exception handling](#)):

EH_ZERO_DIVIDE	EXCEPTION_INT_DIVIDE_BY_ZERO
EH_SINGLE_STEP	EXCEPTION_SINGLE_STEP
EH_NMI	EXCEPTION_ILLEGAL_INSTRUCTION
EH_DEBUG_TRAP	EXCEPTION_BREAKPOINT
EH_OVERFLOW	EXCEPTION_INT_OVERFLOW
EH_ARRAY_BOUNDS	EXCEPTION_ARRAY_BOUNDS_EXCEEDED
EH_INVALID_OPCODE	EXCEPTION_ILLEGAL_INSTRUCTION
EH_DEVICE_NOT_PRESENT	EXCEPTION_ILLEGAL_INSTRUCTION

EH_DOUBLE_FAULT	EXCEPTION_ILLEGAL_INSTRUCTION
EH_DEVICE_ERROR	EXCEPTION_ILLEGAL_INSTRUCTION
EH_INVALID_TSS	EXCEPTION_ILLEGAL_INSTRUCTION
EH_SEGMENT_NOT_PRESENT	EXCEPTION_ACCESS_VIOLATION
EH_STACK_FAULT	EXCEPTION_STACK_OVERFLOW
EH_GENERAL_PROTECTION	EXCEPTION_ACCESS_VIOLATION
EH_PAGE_FAULT	EXCEPTION_ACCESS_VIOLATION
EH_DEVICE_ERROR1	EXCEPTION_ILLEGAL_INSTRUCTION
EH_ALIGNMENT_CHECK	EXCEPTION_DATATYPE_MISALIGNMENT

Project settings

INtime requires some specific project settings in Microsoft Visual Studio for the production of an INtime executable program. The following traverses sequentially through the Project Settings elements, first in the Compiler page, then the Linker page (settings that are not specific for INtime are omitted).

- Compiler Page, Category: General
Debug info: C7 compatible
Preprocessor definitions: at least _WIN32
- Compiler Page, Category: General
To use SEH or C++ Exception Handling, enable exception handling.
- Compiler Page, Category: Optimizations
Optimizations are typically disabled in the debug configuration.
- Compiler Page, Category: Preprocessor
Additional include directories: <INtime>\rt\include (where <INtime> stands for the directory where you installed INtime, usually C:\Program Files\INtime)
Ignore standard include paths: Yes
- Link Page, Common options
/subsystem:console
/HEAP:1048576 (sets the maximum memory pool, here to 1MB)
- Link Page, Category: General
Object / library modules: iwin32.lib rt.lib pcibus.lib netiff3m.lib ciff3m.lib (you can omit pcibus.lib if you do not use PCI functions, and you can omit netiff3m.lib if you do not use networking functions)
Ignore all default libraries: Yes
- Link Page, Category: Debug
Debug info: Microsoft format
- Link Page, Category: Input
Additional library path: <INtime>\rt\lib (w here <INtime> stands for the directory where you installed INtime, usually C:\Program Files\INtime)
- Link Page, Category: Output
Major: 21076, minor: 21076

API mixing

The two APIs each implement a separate set of object types. Although it should be perfectly feasible to code an application exclusively with one of the two APIs, there may be different processes in the system at the same time, that use different APIs. What happens if such processes need to communicate?

- An **iWin32 process** can use the INtime API without limitations; that means it can create INtime objects, lookup tokens, catalog tokens, send and receive units and messages and perform various wait operations. For INtime functions and types, you should use the iwin32rt.h include file instead of the standard rt.h file.
- An **INtime process** can use only a small subset of the new APIs; it can not create iwin32 objects, can not wait for iwin32 objects and can not use Structured Exception handling or C++ Exception Handling. The function definition will indicate whether a particular function can be used in a pure INtime thread. For INtime functions and types, you should use the iwin32rt.h include file instead of the standard rt.h file.

Iwin32x functions

For the communication between processes in the Windows environment and iwin32 processes in INtime, a set of functions is available named 'iwin32x'. These functions can only be used from C or C++ programs. The API provides a subset of the iwin32 API (for obvious reasons, a number of iwin32 functions is not available here) and also includes specific functions, for example to find an INtime node.

The iwin32x functions include:

Event handling	RtCreateEvent
	RtOpenEvent
	RtPulseEvent
	RtResetEvent
	RtSetEvent
Mutexes	RtCreateMutex
	RtOpenMutex
	RtReleaseMutex
Process handling	RtCreateProcess
	RtGetExitCodeProcess
	RtOpenProcess
	RtTerminateProcess
Semaphores	RtCreateSemaphore
	RtOpenSemaphore
	RtReleaseSemaphore
Shared memory	RtCreateSharedMemory
	RtOpenSharedMemory
General purposes	RtCloseHandle
	RtImportHandle
	RtSetNode
	RtWaitForSingleObject

Priorities

INtime supports three thread priority ranges. Internally the INtime range is always used: from 0 to 254, zero being the most important. Priorities between 0 and 127 influence interrupt enabling.

iWin32 (functions such as GetThreadPriority, SetThreadPriority) has a priority range from -15 to +15, the numerically highest value is the best. These values are mapped to the INtime priorities as follows:

<u>iwin32 priority</u>	<u>INtime priority</u>	<u>Win32 priority name</u>
-15	253	THREAD_PRIORITY_IDLE
-14	249	
-13	245	
-12	241	
-11	237	
-10	233	
-9	229	
-8	225	
-7	221	
-6	217	
-5	213	
-4	209	
-3	205	
-2	201	THREAD_PRIORITY_LOWEST
-1	197	THREAD_PRIORITY_BELOW_NORMAL
0	193	THREAD_PRIORITY_NORMAL
1	189	THREAD_PRIORITY_ABOVE_NORMAL
2	185	THREAD_PRIORITY_HIGHEST
3	181	
4	177	
5	173	
6	169	
7	165	
8	161	
9	157	

10	153	
11	149	
12	145	
13	141	
14	137	
15	133	THREAD_PRIORITY_TIME_CRITICAL

iWin32 does not support priority classes.

RTX thread priorities (RtSetThreadPriority, [RtAttachInterruptVector](#) etc) range from 0 to 127, with 127 being the best priority. An RTX priority is translated into an INtime priority as follows: $\text{prio(INtime)} = 253 - \text{prio(RTX)}$.