

# **Choosing INtime over RTX**

## **A Comparison of Real-Time Extensions**

August, 2005



This document is protected by US and international copyright laws.

INtime and iRMX are registered trademarks of TenAsys Corporation.

All trademarks, registered trademarks and brand names are the property of their respective owners.

Other companies, products, and brands mentioned herein may be trademarks of other owners.

Information regarding products other than those from TenAsys has been compiled from available manufacturers' material. TenAsys cannot be held responsible for inaccuracies in such material.

TenAsys makes no warranty for the correctness or for the use of this information, and assumes no liability for direct or indirect damages of any kind arising from the information contained herewith, technical interpretation or technical explanations, for typographical or printing errors, or for any subsequent changes in this article.

TenAsys reserves the right to make changes to specifications and product descriptions at any time, without notice, and without incurring any liability. Contact your local TenAsys sales office or distributor to obtain the latest specifications and product descriptions.

Copyright © 2005, TenAsys Corporation, All Rights Reserved

No part of this guide may be copied, duplicated, reprinted, and stored in a retrieval system by any means, mechanical or electronic, without the written permission of the copyright owner.

August, 2005 Edition

# Choosing INtime over RTX

## A Comparison of Real-Time Extensions

### *Table of Contents*

<b>Executive Summary .....</b>	<b>2</b>
<b>Introduction .....</b>	<b>3</b>
Is a real-time extension for Windows really necessary?	3
Doesn't a faster processor make Windows more deterministic?	3
<b>Highlighted Comparison.....</b>	<b>4</b>
With so many similarities, where are the differences?	4
<b>Safe, Secure, Reliable .....</b>	<b>5</b>
Safety in a Proven History	5
Continuing the legacy	6
Searching for a legacy	6
Security in Architecture	8
Encapsulation of Windows	8
Dependence upon Windows	9
DSM Role in Reliability	13
What is the DSM?	14
Building robust complex real-time systems	14
<b>Conclusion .....</b>	<b>15</b>

## Executive Summary

The INtime® real-time extension for Microsoft® Windows® is the only fully-protected solution available for building hard real-time applications which run concurrently with the Windows NT, 2000, and XP operating systems on a single hardware platform. Based on over 20 years of reliable field-proven iRMX® technology, INtime combines robust industrial-grade control with enterprise-rich Windows functionality so that your applications can take full advantage of every Windows feature and thousands of off-the-shelf applications, without having to sacrifice real-time responsiveness. Using TenAsys software to implement your real-time Windows applications reduces software complexity, removes redundant hardware cost, and speeds time to market.

TenAsys software helps you achieve these goals because:

- The unique architectural design of INtime, which encapsulates the entire Windows operating system as a separate and isolated task, insures safe implementations and minimizes exposure to unknown interactions between real-time elements and the Windows kernel and subsystems.
- Our commitment to provide only real-time solutions for the x86 embedded market focuses all of our attentions and energies where it matters most, on your success.
- Scaleable INtime product solutions mean your real-time application binaries can be deployed locally, on a single Windows PC, or across multiple stand-alone embedded PCs, hosted by a network-attached supervisory Windows PC, without the need to rewrite or even recompile your carefully-tuned and tested real-time code.
- Preservation of investment can be accomplished by utilizing one or more of the built-in INtime APIs, either separately or combined. In addition to the native INtime API, we include APIs to facilitate the reuse of existing iRMX and RTX real-time applications, usually with only minor modifications and a recompile of existing code.
- Close ties with and close proximity to the Microsoft operating system development teams means fast response and early access to key Microsoft Windows technologies for the TenAsys development team. Every version of Windows NT/2000/XP is supported by INtime, including Windows Server 2003.
- There is absolutely no question regarding the long-term viability of the INtime kernel and its supporting code-base. INtime is based directly on the proven history (greater than 20 years) and extensive application base of the iRMX real-time operating system, originally developed by Intel and now exclusively owned and supported by TenAsys.

When using Windows as a platform for embedded real-time systems, it is critical that the real-time extension be reliable and deterministic, both now and in the future. TenAsys has made reliable real-time Windows a reality by utilizing proven real-time technology that integrates seamlessly with Windows. Real-time Windows applications based on TenAsys software have been deployed ranging from data communication equipment and medical imaging applications to factory floor automation.

Supporting this array of applications is the INtime proven and protected mode programming model, which is a must for any real-time application to be considered safe, secure, and reliable. Just as your Windows applications execute in user mode, your real-time applications should also execute in user mode. Competing solutions are built to have your real-time applications run completely unprotected inside the Windows kernel, where programming errors that result in bad pointers, stack overruns, page faults, etc. result in compromising both the real-time and the Windows environment. This is never the case if your real-time application is built on the INtime protected system.

## Introduction

There are several vendors providing products today which are commonly referred to as real-time extensions for the Windows NT/2000/XP family of operating systems. Of those products, we are often asked to compare INtime with RTX from Ardence (formerly VenturCom). While the primary end goal of both products is the same, to provide a hard real-time environment for the Windows platform, the architectures and product capabilities are quite different. Given a thorough understanding of the differences between these two products, we believe you will find INtime to be the obvious and superior choice for your real-time Windows applications.

### **Is a real-time extension for Windows really necessary?**

The deterministic nature of a real-time system forces a unique set of requirements upon software applications. A simple definition of a real-time system is one in which the time required to respond to an event is just as important as the logical correctness of that response. Hard real-time systems require the highest degree of determinism and performance. Frequently, their worst case event response requirements are measured in tens of microseconds.

Bounded response to events is the key to defining a hard real-time system. Real-time systems require determinism to insure predictable behavior of the system. Without determinism a system cannot be called real-time. Without bounded determinism a system cannot be classified as hard real-time.

The level of determinism required is a function of the frequency of the real-time events in the system (length of the time interval between events) and the effect of delays on the dynamics of that system. That is, how frequently do events occur and how quick and repeatable must the system be in response to those events. The ability to place a finite and acceptable bound on the value of these numbers is what distinguishes a hard real-time system from other systems.

### **Doesn't a faster processor make Windows more deterministic?**

Faster processors, memory, and peripherals improve the aggregate performance of a system, but they generally do not directly affect the bounded determinism of a system. The worst-case response time to an event will not be significantly changed by using a faster processor; increased speed may decrease the spread and intensity of the variations in response to an event, the jitter, but it will not eliminate jitter, especially worst-case jitter.

Improving the performance (or speed) of a system is, however, very useful. More performance increases the complexity of the algorithms one can implement in a given period of time. Therefore, the quality of the control and data acquisition systems that can be implemented in software are improved by using a faster system. However, bounded determinism is still required to insure that a stable, accurate, and predictable system, regardless of the performance level of that system, can always be deployed.

Yes, real-time extensions are necessary in order to make hard real-time Windows applications stable and predictable. The only viable alternative would be to employ dedicated hardware, which increases system complexity and cost, and boxes you into a solution that fits today's parameters but frequently proves too inflexible to address tomorrow's added twists.

## Highlighted Comparison

Despite the common objectives of INtime and RTX to insure the real-time responsiveness of the Windows platform, the differences between these two products and companies could not be more profound. Before highlighting those differences it is worth quickly noting the similarities of the two products.

Both products are a “real-time extension to Windows” enabling developers to create hard real-time applications for the Windows platform. In both cases, real-time Windows applications must be divided into two basic parts: deterministic and non-deterministic. The non-deterministic part(s) execute in the Windows environment, and the deterministic part(s) execute in the real-time extension’s environment. In order for these two basic parts to work together as a single application, managed access to a variety of shared objects (mailboxes, semaphores, mutexes, memory, etc.) must be provided by the real-time extension.

INtime and RTX are both able to leverage the Windows platform and Microsoft’s substantial suite of development tools to speed time to market. Additionally, both products provide direct access to I/O and memory in the real-time environment, a fixed priority scheduling system with priority-inversion protection, and simplified interrupt-handling services. All of these features allow developers to create and deploy real-time applications without the need to write complex and cumbersome device drivers for access to real-time hardware. These features greatly simplify the development and deployment of control and data acquisition algorithms in the real-time environment.

Finally, both products provide access to high-speed interval timers for accurate, low-drift time measurements and for generating accurate periodic events to insure precise control of real-time systems. The accuracy and drift of these timer elements is a strong function of the specific platform, being optimal on APIC uniprocessor systems and multi-processor systems.

	TenAsys – INtime	Ardence – RTX
<b>deterministic hard real-time extension for Windows</b>	Yes	Yes
<b>real-time scheduler with multiple fixed priority levels</b>	Yes	Yes
<b>priority-Inversion protection</b>	Yes	Yes
<b>direct access to I/O and physical memory</b>	Yes	Yes
<b>leverage Microsoft development tools</b>	Yes	Yes
<b>high-speed interval timers</b>	Yes	Yes
<b>simplified interrupt handlers</b>	Yes	Yes
<b>multi-processor support</b>	Yes	Yes

Table 1: INtime and RTX product similarities

### With so many similarities, where are the differences?

A simple checklist of features, like that shown in the table above, is inadequate to compare the two products. There are some very significant and deep architectural differences between these two products which affect the usefulness and reliability of such a feature list. There are also many subtle differences which may not become apparent to the developer until well down the development path.

This paper describes the key differences between INtime and RTX and explains how those differences impact your real-time Windows applications. These differences affect not only the application development process but also your deployment options and the support efforts required to maintain your real-time applications in the field. We are confident that upon review of these differences you will find

TenAsys to be your trusted and only choice as the long-life supplier of real-time technology on the x86 platform.

A summary of some of the key differences between INtime and RTX are shown below:

	TenAsys - INtime	Ardence - RTX
<b>Real-time kernel legacy</b>	Based on iRMX, with 20+ years of severe, mission critical applications	Relatively new device driver scheduler
<b>Real-time kernel environment</b>	Separate hardware task protected from Windows	Inside Windows kernel
<b>Stand-alone binary-compatible remote real-time nodes</b>	Yes	No
<b>Real-time applications run in a protected address space</b>	Yes	No
<b>Process and thread execution mode</b>	User mode - ring 3	Kernel mode - ring 0
<b>Processes and threads protected from Windows</b>	Yes	No
<b>Processes and threads protected from each other</b>	Yes	No
<b>Real-time debug model</b>	One environment, debug and deploy on the real-time kernel	Debug as a Windows application, deploy on the real-time kernel
<b>API</b>	INtime, iRMX, and RTX	RTX
<b>Core business</b>	x86 real-time operating systems	Distribution sales, consulting, software development

Table 2: Key differences between INtime and RTX

## Safe, Secure, Reliable

Every developer endeavors to build applications that can be described by these three adjectives. When considering the nature of real-time systems, which are routinely intended to operate continuously and with minimal user intervention, these three adjectives are doubly important. Without a safe, secure, and reliable foundation on which to build your application it is very difficult to achieve a safe, secure, and reliable result.

## Safety in a Proven History

A safe system is one on which you can depend. Safety is best insured by building upon a proven foundation. INtime is derived directly from the iRMX kernel, originally developed and introduced in 1980 by Intel for their x86 architecture.

Maintained, supported, and enhanced by engineers that were part of the original Intel iRMX development team, INtime builds upon a solid history of use, application, and experience. TenAsys' development and support staff have been part of the original iRMX 86 and INtime development process since the very beginning (joining between the late 70s through the early 90s), either as members of the original iRMX and INtime engineering teams or as iRMX and INtime application developers. When the iRMX product family was transferred to RadiSys, as part of RadiSys' purchase of the Intel Multibus division in the mid 90s, those engineers moved along with the real-time software products team, continuing to develop, refine, and support iRMX and its progeny, INtime. During 2000, that team left RadiSys to form TenAsys, in order to concentrate all of their energies on the x86 real-time market, and have become the sole licensee, developer, and supporter of all iRMX and INtime real-time products.



iRMX is the gold standard in real-time software for the x86 architecture, borne out by its proven and current use in thousands of demanding real-time applications worldwide. This versatile RTOS has established its capabilities many times over from applications that utilize small-footprint, kernel-only solutions to those that require a highly-configurable, full-service, hard real-time operating system.

### Continuing the legacy

INtime continues to build upon the iRMX legacy by providing equally safe, secure, and reliable operation for real-time applications that demand the best combination of real-time and Windows. INtime leverages that proven technology by integrating it closely with Windows. As a closely-linked product family, INtime and iRMX bring an unprecedented level of reliability and safety to hundreds of critical real-time applications, including:

- Industrial control and robotics
- Air and ground traffic control
- Medical imaging
- Test and measurement
- CNC machining
- Material handling
- Process control
- Radar and avionics
- Simulation
- Mail and check sorting

### Searching for a legacy

Unfortunately, one cannot make similar claims regarding the legacy of RTX. In relative terms it is a new product, built by an intermittent development team. Contrasted with the coherent and continuous development process that is a basis of the INtime legacy; a disconnected process results in the loss of valuable information, impeding the creation

of the critical knowledge base that is essential to service, support, and respond to the requirements of demanding real-time applications and customers.

RTX has only scratched the surface of real-world applications. Regardless of the competency of the Ardence developers, real-time kernels and their associated scheduling, interrupt handling, and other mechanisms are complex. These systems require years of field testing before a stable reliable solution is available. Users of Ardence's RTX have experienced a number of bugs and "enhancements" which are a reflection of the long process required to build a safe kernel. This was illustrated most recently with the addition of a "deterministic memory" feature to RTX version 6.0 (released March, 2004).

### **Comparison: “deterministic memory”**

“Deterministic memory,” the term used by Ardence to describe a recently added “feature” for RTX should really be viewed as a long overdue bug fix to the core RTX memory allocation implementation. It refers to is an assurance that requests to allocate memory by real-time applications, when making direct real-time API calls or indirect C library calls, will be bounded in time. This is accomplished by limiting memory allocation to a pre-allocated memory pool instead of relying on Windows to perform memory allocation for the real-time environment.

This illustrates the troubles associated with building real-time applications on kernels that have not been sufficiently field tested. It also highlights the problems associated with the architectural approach of the Ardence solution, which will be discussed later in this paper.

The INtime solution has, from the very beginning, managed all real-time memory from a pre-allocated, non-paged, memory pool. In fact, there is no option on this point, all memory managed by the INtime kernel for use by real-time applications is guaranteed to come from a static, non-paged, memory pool without any interference or involvement by Windows.

A subtlety associated with the above RTX “feature” materializes when real-time threads attempt to allocate more memory than has been pre-allocated in the “deterministic memory” pool by the RTX kernel. Attempts to request more memory than has been pre-allocated are satisfied by requesting additional memory from Windows. Unfortunately, this results in the very same latent determinism error which should have been fixed by this new “feature.” Because it is difficult in some applications to conclusively determine if all memory allocations will fit within the pre-allocated pool, due to indirect memory allocations that occur in libraries, etc., this “feature” returns one to the non-deterministic problem that prompted its addition in the first place!

In an INtime system, attempts to allocate more memory than that which has been pre-allocated by the kernel (a user-adjustable parameter) will result in a hard program error (e.g., an error returned by a function call that needs to allocate memory). In other words, if your real-time application will be using more memory than you have asked the INtime kernel to pre-allocate, you will be overtly informed of that fact. You will not be subjected to the vagaries of latent determinism issues which can show up as unexplainable and unpredictable behavior of your real-time application. With an overt notification the problem is easily identified and fixed.

Because INtime pre-allocates *all* of its memory from Windows (at boot time), and thenceforth manages that entire block internally in a separate, distinct, and isolated hardware task, the only fragmentation or allocation concerns are those internal to the real-time applications and threads; Windows, its drivers, and its applications have zero impact on the fragmentation or determinism of real-time memory. Real-time threads can allocate and free this memory as required, greatly simplifying the implementation of your application and further insuring that hard real-time determinism is achieved.

By using the words “deterministic memory” to describe this new “feature” one has to ask if any RTX applications written prior to version 6.0 could ever have been classified as deterministic! Certainly, it would be hard to insure that they qualify as hard real-time applications under all operating conditions. Given the subtleties of the new behavior of this “deterministic memory” it is questionable that even RTX 6.0 applications can always be referred to as hard real-time applications. What other elements of the RTX subsystem are awaiting “improvements” in order to insure proper isolation and deterministic behavior?

## Large memory allocations

For those real-time applications that require unusually large arrays of physical memory (many tens of megabytes or even gigabytes of RAM), such as image processing applications, INtime can be configured to isolate and reserve very large segments of physical memory for exclusive use by real-time processes. In this case INtime converts the RAM into “non-Windows memory” or memory of which Windows has no awareness. This real-time memory is completely isolated from the Windows kernel, drivers, and applications. This technique is an alternative to utilizing the memory that the INtime kernel normally reserves from the Windows non-paged memory pool.

“Non-Windows memory” is standard RAM that is isolated and reserved by the INtime system before Windows makes its own memory allocations. It is reserved for exclusive use by real-time processes and threads; there are no special hardware tricks required to make it work. Real-time applications built on INtime have access to such features because the INtime system comprises a complete real-time operating system that executes independently of the Windows system. This method of isolation is unique to INtime and is one of the significant differences between INtime and RTX.

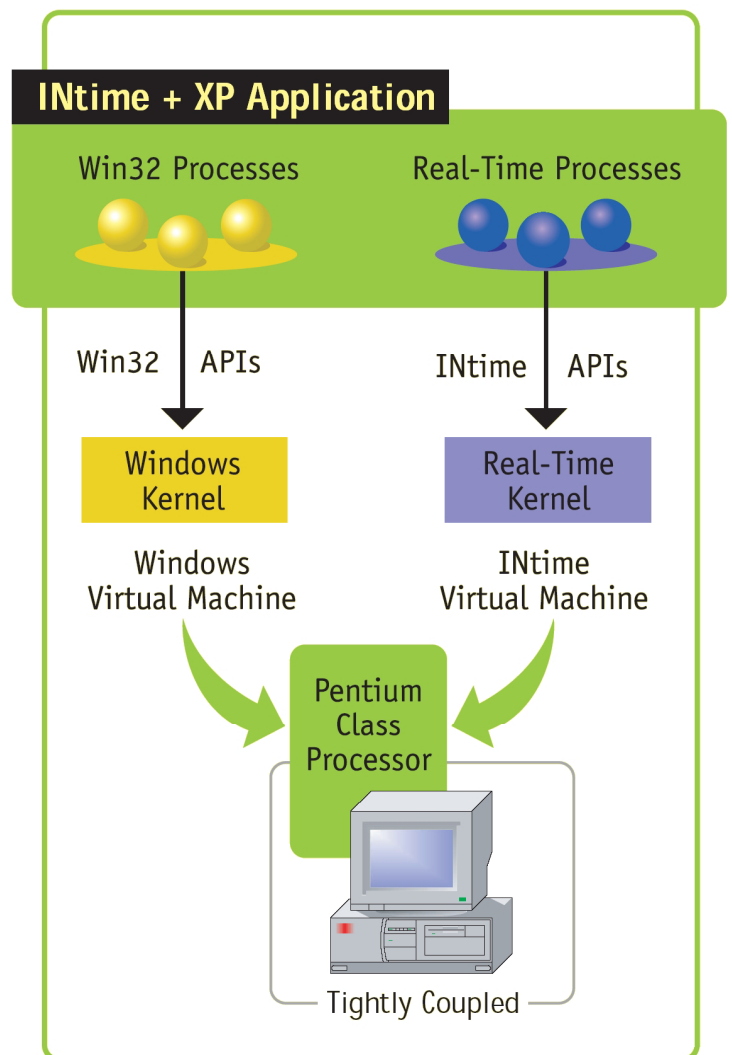
## Security in Architecture

Any solution that claims to provide hard real-time capabilities for a Windows system must provide an alternate kernel to perform the scheduling and execution of the real-time processes and threads. In fact, every hard real-time solution available for Windows today uses such an approach. In all of these systems a real-time kernel runs on the same hardware as the standard Windows kernel, sharing the CPU, memory, and I/O resources.

The architecture of a real-time Windows solution has a strong bearing on the security of the system. Intel’s 32-bit x86 processors (80386 through modern Pentiums) include unique hardware features that accommodate building systems that can run simultaneous multiple operating systems. INtime takes full advantage of these features to build a secure environment in which real-time processes and Windows processes are isolated and separate on the same machine.

## Encapsulation of Windows

In an INtime system a separate “hardware task environment” is fashioned to contain the real-time kernel and all of its processes, including code, variables, I/O, and interrupts. This hardware task environment is constructed by building completely separate GDT, IDT, page tables, and other data structures, which are specific to the operating system and independent of those maintained by Windows. Finally, the entire



Windows environment is “encapsulated” by the INtime real-time kernel, setting it up as the lowest-priority task in the real-time environment.

According to the *Computer Desktop Encyclopedia*, “encapsulation means to make the data and processing within the object [Windows] private. This allows the internal implementation of the object [Windows] to be modified without requiring any change to the application [INtime] that uses it.” In this sense, Windows is the object and INtime is its container. Encapsulation has the added advantage of greatly simplifying compatibility with future releases of the Windows operating system.

This isolation between INtime and Windows is significant, as it insures a secure operating environment for real-time applications in a Windows environment; the only “hooks” into Windows required by the INtime kernel are located in the Windows HAL, which is “hooked” to insure that shared hardware is properly managed between the two operating systems. Specifically to:

- Intercept attempts to modify the system clock rate, insuring that the INtime kernel controls the system time base.
- Manage those resources exclusive to INtime or shared with Windows.
- Insure that interrupts reserved for real-time use are never masked or assigned for use by any Windows software.

The full suite of standard Microsoft Windows HALs are recognized and supported by INtime software, including support for single-CPU systems with a multi-processor HAL. Only vendor-specific HALs developed for older non-standard hardware are not supported by INtime. Virtually all hardware systems sold and in use today utilize the Microsoft standard HALs.

### **Dependence upon Windows**

Compare the independent method of encapsulation used by INtime to the dependent driver approach used by RTX. TenAsys’ approach to integrating a real-time kernel with Windows is to encapsulate the Windows environment and provide an independent and complete real-time operating system environment in which all real-time processes and threads execute. Ardence’ approach is to insert their real-time kernel directly inside the Windows system as a Windows driver, making it dependent upon the vagaries and whims of the entire Windows system.

As a Windows driver, the RTX kernel and its real-time applications execute in kernel mode (ring 0). Standard Windows applications always execute in user mode (ring 3) preventing them from making modifications (either deliberate or unintended) to the code and data structures belonging to other Windows applications (ring 3), to the Windows kernel (ring 0), or to real-time code (ring 0).

The reverse is not true. Real-time RTX threads can contribute to instability of the entire system because they operate in ring 0, as part of the Windows kernel. There is no protection for the system when real-time RTX applications go astray. Any bug in an RTX real-time thread that “crashes” the real-time application also crashes the Windows system. This is further compounded by the fact that debugging ring 0 code is an inefficient and time consuming process.

### **Ring 0 versus ring 3**

Crashing an INtime application does not crash Windows because all INtime processes operate as user level (ring 3) threads, exactly like standard Windows applications; they can be stopped or restarted without destabilizing the rest of the system. This is a significant advantage when debugging and testing real-time applications.

All of the hardware protection mechanisms provided by the CPU to maintain isolation between individual applications and between applications and the kernel are utilized, with no I/O access limitations. Unlike Windows applications, INtime applications *can* perform direct I/O to ports and memory because INtime applications are allowed to execute the CPU's IN and OUT instructions and can "map" physical addresses into their virtual address space. INtime threads are also not restricted from manipulating the processor's interrupt flag via the STI and CLI instructions. The result is a flexible but secure and controlled run-time environment where problems within the real-time system can be easily isolated, and are not liable to create problems elsewhere in the system. Executing real-time threads at user level (ring 3) also greatly simplifies the debugging process.

There is no performance penalty for executing real-time applications at the more reliable and protected ring 3 operating level. Software executes at exactly the same speed in ring 3 as it does in ring 0. The differences between these two operating levels have only to do with the execution of certain privileged instructions, fault (trap) handling, memory protection, and the use of debugging tools.

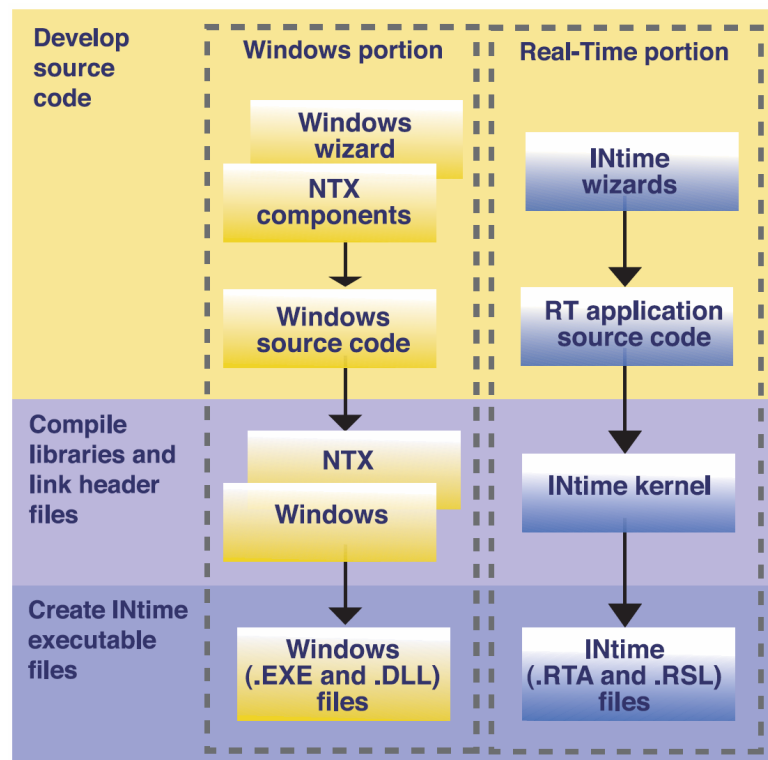
The advantages of application debugging compared to kernel debugging have much to do with efficiency and convenience:

- Kernel debugging frequently requires the use of "remote debugging," using a second machine, meaning it is not easily performed in a self-hosted environment.
- Kernel debugging essentially puts the entire machine into debug mode, making it very difficult to multi-task on the machine while debugging.
- Crashing a kernel thread requires a reboot of the entire machine, whereas crashing an application thread only requires that one kill the offending process and start over.

In an RTX system, in order to debug and test a real-time application without the impediment of locking up the machine in the event of faults and process lock-ups you must perform your debugging in a "simulation environment." That is, real-time RTX applications must first be compiled to execute on the standard Windows kernel and, therefore, *do not* execute in real-time on the real-time RTX kernel. Debugging on the real-time RTX kernel requires use of a kernel debugger, with greatly reduced system functionality.

INtime applications, on the other hand, are developed, debugged, and tested entirely within the target environment. There is no intermediate simulation step required to debug INtime real-time code. INtime applications always execute on the real-time kernel and always run in real-time. Debug and test is performed directly within the Microsoft Visual

Studio .NET debugger, in real-time. There are no restrictions associated with testing and debugging real-time applications like those associated with using a kernel debugger.



Attempting to verify ring 0 code in kernel mode with the RTX debugger results in restrictions like the following:

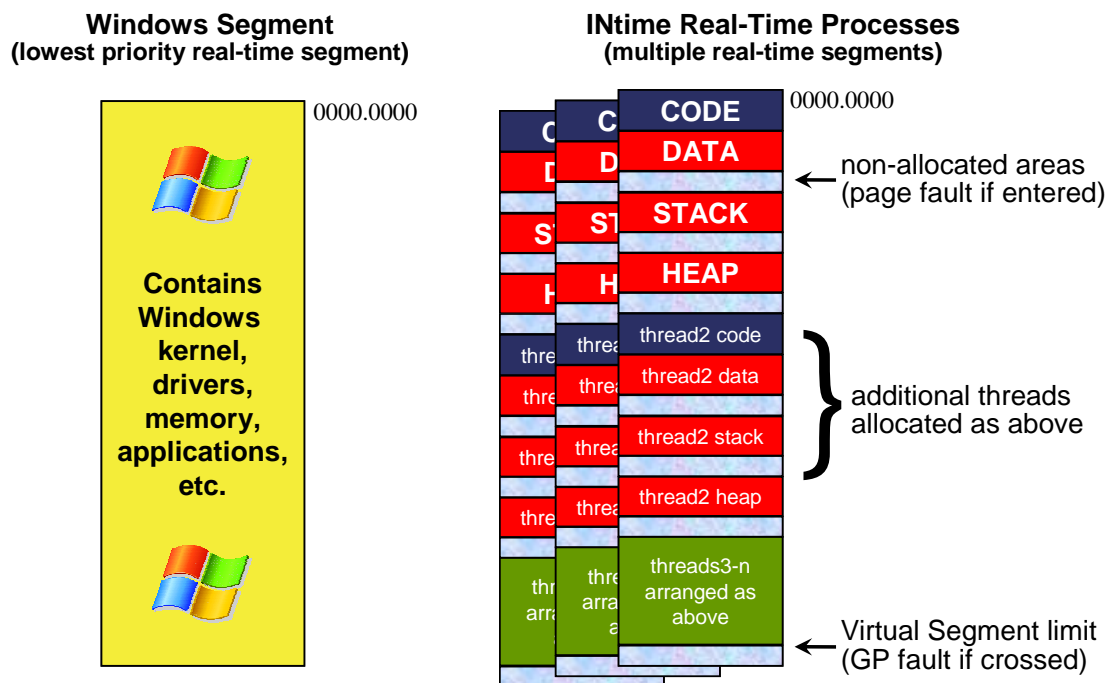
- Disassembly or single step of RTX system calls is not allowed.
- Deterministic behavior does not exist when using the RTX debugger.
- The RTX debugger cannot be used to debug RT-TCP/IP Applications.
- The RTX debugger cannot be used to debug applications on a remote computer.

None of the restrictions above apply to testing and debugging INtime applications on the INtime real-time kernel; whether operating INtime as a local Windows real-time system or as a stand-alone remote real-time node.

### Memory management

Software that executes in ring 0, like the RTX kernel and its real-time tasks, can access the entire memory space of the system, with no restrictions. This means that applications, the Windows kernel and device drivers, and other real-time threads can have their memory overwritten by an errant real-time thread. There is no hardware-enforced address isolation or memory protection available to software that runs in ring 0. Unfortunately, these types of programming errors are very difficult to detect, and can appear as latent failures once a real-time application has been deployed.

The INtime kernel places its real-time applications into multiple distinct protected memory environments that are separate from those created by Windows. The real-time processes and threads reside within non-paged ring 3 virtual memory segments. This address isolation exists not only between real-time processes and Windows processes but also between multiple real-time processes. It is setup and configured by the INtime kernel and enforced by hardware in the CPU. No special tools or programming procedures are required of the application developer to take advantage of this feature; the memory protection is supported automatically using standard Windows development tools.



Within an RTX system the Windows kernel, drivers, and the RTX kernel and its real-time applications all reside in a single ring 0 address space. There is no process-to-process memory protection like that found in the INtime system or like that enforced by Windows between Windows applications.

Note that the term “virtual memory segments” used to describe the INtime process management scheme in the figure above does not mean that real-time memory segments are paged. Whenever there is a switch to an INtime real-time task or thread (e.g., as the result of a real-time event or timer activity), real-time code is guaranteed to execute immediately from physical memory. There is never any paging of the real-time kernel or any of its processes and threads to off-line storage. All real-time code elements are guaranteed to be resident in non-paged physical memory at all times. Without this basic system rule in place there would be no way to insure that the INtime kernel and its applications operate deterministically.

By taking full advantage of the virtual addressing hardware built into the CPU, the following protection and debugging mechanisms are provided within the INtime real-time environment:

- code, data, stack, and heap are in separate, non-contiguous pages
- code is placed in read-only pages for extra protection
- paging is utilized for address isolation and pointer overrun protection

Every real-time thread is assigned distinct memory pages for its CODE, DATA, STACK, and HEAP memory regions. These pages are separated by non-allocated pages of memory so that pointer overruns can be easily and quickly detected. CODE pages are further protected by marking them as read-only.

A key advantage to the INtime kernel’s encapsulation approach is the numerous memory protection mechanisms employed to insure secure operation and to simplify debugging and testing real-time applications. This is a stark contrast with the RTX solution which requires real-time threads to run within the Windows kernel at ring 0, without any address isolation or memory protection between multiple real-time processes or between real-time processes and Windows processes.

### **Windows STOP (“blue screen of death”)**

Because the RTX run-time environment is dependent upon the Windows kernel for several key features (esp. memory allocation) it is very difficult to write real-time applications that accommodate the undesirable but inevitable blue screen of death (BSOD) scenario. For example, care must be taken when writing RTX applications so they can work in the context of a Windows BSOD:

- RTX memory allocation functions will not work, such as `calloc()` and `malloc()`.
- Creating RTX objects will not work; including creation of threads, timers, objects, and variables.

With such limitations, an RTX application must include liberal doses of conditional code like the following:

```
if (bBlueScreen==TRUE)
    Sleep(INFINITE);          // no destructors or process exits allowed
```

This needs to be done in every location that might involve a memory de-allocation or process exit, in order to avoid locking up the entire RTX system following a BSOD. In other words, you must explicitly plan and code for the rare, but inevitable exception of a Windows blue screen throughout your real-time code, which, for simplicity sake, should be completely independent of Windows.

The complete stand-alone RTOS on which INtime applications execute handles the BSOD condition transparently, the INtime kernel and its processes will continue to operate indefinitely. The only planning required is in the obvious locations, where your real-time code depends upon objects which are shared with Windows, such as a semaphore or mailbox. Otherwise, you do not need to worry about the BSOD!

The INtime system provides a STOP manager specifically designed to intercept any Windows STOP event. Real-time applications are provided with notification of a BSOD so they may take appropriate actions to deal with such an event. Generally, detection of a BSOD by the INtime STOP manager can be used by the real-time application to gracefully shutdown or restart the system, or to send a signal to some other system via a real-time controlled communication interface (e.g., serial or Ethernet) indicating that action or attention is required.

There are a few INtime C library calls (specifically, those that deal with file I/O) that can be affected by a Windows BSOD. In this case, any open resources are closed and subsequent calls to these functions by real-time applications result in standard C library error codes being returned (such as file not available); your real-time threads will not be suspended as a result of making these calls. There is no reason to be concerned about accessing specific functions in the C library, nor is there any reason to be concerned about memory allocation calls. Likewise, there is no reason to stop creating threads or other objects, as these are handled separately from Windows.

Real-time applications in an INtime system never make calls to Windows, they only make calls to the INtime RTOS kernel and its libraries. Within the INtime environment the Windows operating system exists only as the low-level thread (the idle task). Further proof of the superiority of encapsulating Windows versus integrating an RTOS within the Windows kernel.

## **DSM Role in Reliability**

Complex real-time tasks, especially those that may be distributed over multiple hardware elements, require solutions that go beyond simply providing a hard real-time execution environment. Insuring integrity in such systems necessitates the use of a system manager that can notify processes and threads of the existence or disappearance of a key resource.

The INtime execution environment is not limited to providing real-time services to a single Windows machine. The INtime kernel (and its real-time applications) can be installed and run on a standalone embedded platform, connected to a Windows workstation and operating as remote nodes. Remote INtime nodes are distinct independent hardware and software modules (CPU, memory, I/O) with a communication link to a Windows host that may optionally be hosting a local INtime real-time environment.

Communication between a Windows application and a remote INtime node utilizes the same API as that used by a local INtime application and Windows, except Ethernet or a serial link is used as the communications medium, rather than memory. All the API calls used to communicate between Windows and real-time threads are available and work identically in this configuration, only shared memory is unique to the local configuration (INtime sharing a PC with Windows).

INtime remote nodes are the mechanism by which real-time Windows applications can be scaled from a system that incorporates a single shared hardware module (Windows+INtime) to a system that incorporates multiple hardware modules (Windows+INtime+INtime+...). This scalable architecture can be implemented without the need to rewrite software, or even recompile! The INtime Distributed System Manager (DSM) is key to providing this flexibility.

## What is the DSM?

The Distributed System Manager (DSM) is a cooperative, multiple process application that manages local and distributed INtime systems. It tracks the state of the system, monitors the health of its components, and cleans up in the event of a tracked component's termination or failure, including the Windows operating system and applications.

The DSM was originally conceived as a service for distributed INtime systems which contain multiple, remote, standalone, real-time nodes that communicate with a Windows host via either Ethernet, serial, or backplane. However, it is a very powerful tool that can also be used for the simpler and more common real-time system where a local INtime node must monitor the state of the Windows operating system, its processes and threads, and the real-time objects shared with those Windows processes and threads, such as mailboxes and semaphores. In this case, the DSM provides a clean, simple, and reliable mechanism which can be used to inform real-time processes and threads that the Windows system is no longer available and the appropriate action should be taken.

The INtime DSM performs these key tasks:

- tracks dependencies between Windows processes and threads and real-time processes, threads, and objects
- provides notification when tracked processes, threads, and objects terminate or fail
- cleans up shared resources when a tracked Windows process or thread, or the entire Windows environment, terminates or fails

The DSM consists of two parts, a Windows DSM process that executes in the Windows environment and a real-time DSM process that executes in the real-time environment. These two processes ("the DSM") cooperate to insure integrity of an INtime system. The DSM monitors processes, threads, and shared resources (or simply, "objects") that have been explicitly declared with dependencies. It continuously checks for the existence of those objects and, if it finds that an object is no longer available, it notifies all interested parties of that event.

When the DSM recognizes that Windows has undergone a catastrophic error (i.e., a blue screen is imminent) a system cleanup begins. In this case, all real-time threads that have been previously declared as dependent on a Windows shutdown are notified, triggering each thread to perform its application-specific procedure to handle this undesirable event. Likewise, any INtime system-specific internal cleanup required to insure continued operation is performed. Primarily, this means the C library is informed so that any calls that would result in a call to the Windows system (such as file I/O) will still work, but may return errors to the calling threads.

Obviously, the DSM is a very powerful mechanism that can be used to build reliable real-time systems that can be scaled from a single hardware platform to one that incorporates multiple real-time modules connected to a supervisory Windows host.

### Building robust complex real-time systems

Use of the DSM is an optional component, which can be built upon to develop more complex and robust systems. Its functionality gives you the ability to build completely standalone applications that can run on a remote real-time node or on a local Windows-hosted system. Such scalable system solutions give you the flexibility to adapt to multiple hardware requirements within a family of products.

Scalability of solutions is achieved by the fact that the INtime and Windows binaries you create for an "INtime local node" will run unmodified on a remote deployment or "INtime remote node." Unlike many

RTOS solutions, INtime remote nodes can be built to support single-chip 386 and 486 systems. Many 32-bit embedded RTOS solutions today are incapable of supporting x86 processors below the Pentium level.

## **Conclusion**

A proven, protected mode programming model is a must for any real-time application to be considered safe, secure, and reliable. Just as your Windows applications execute in user mode, your real-time applications should also execute in user mode. INtime provides that environment, RTX does not. RTX cannot support user mode applications running in real-time, all RTX applications run inside the Windows kernel completely unprotected. In kernel mode, simple programming errors that result in bad pointers, stack overruns, page faults, etc. result in compromising both the real-time and the Windows environment.

When looking to Windows as a platform for embedded real-time systems, it is critical to assure that the real-time extension provides the necessary level of reliability and determinism, both now and in the future. By utilizing proven real-time technology and providing seamless integration with Windows, TenAsys has made reliable real-time Windows a reality. Corporations can utilize Windows for a variety of real-time embedded applications, from data communication equipment and medical imaging applications to factory floor automation. Utilizing INtime, these embedded real-time applications are able to take full advantage of the standard Windows user interface, its powerful network capabilities, rich development tools, and off-the-shelf software, and still deliver the rock solid, reliable performance required of hard real-time systems.