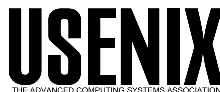
The following paper was originally published in the

Proceedings of the 3rd USENIX Windows NT Symposium

Seattle, Washington, USA, July 12–13, 1999

HARD REAL-TIME WITH RTX ON WINDOWS NT

Mike Cherepov and Chris Jones



© 1999 by The USENIX Association
All Rights Reserved
For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes. This copyright notice must be included in the reproduced paper.
USENIX acknowledges all trademarks herein.

Hard Real-Time With RTX on Windows NT

Mike Cherepov, Chris Jones
{cher, clj}@vci.com
VenturCom, Inc.
Cambridge, MA
www.vci.com

Abstract

For a variety of reasons, Microsoft Windows NT is increasingly being considered as a platform for deployment of real-time systems. In order to meet the stringent latency requirements of hard real-time systems, it is necessary to augment the capabilities of Windows NT. We examine VenturCom's RTX, which provides a real-time subsystem running on Windows NT. It implements deterministic scheduling of real-time threads, inter-process communication mechanisms between the real-time environment and the native NT environment, and other extensions to Windows NT which are often found in specialized real-time operating systems. We discuss how the components of RTX provide these features, explore current results and experiences, and point out possible future directions for enhancement.

1 Introduction

Microsoft Windows NT's popularity and market share have been growing. The reasons for this are varied, including:

- The increasing power and declining price of Windows NT platforms.
- The many applications available on the platform.
- The variety of development tools available on the platform.
- The richness of the Win32 Application Programming Interface (API).
- The large number of developers, support personnel, and end users who are familiar with the system.

Because of the added complexity and cost of maintaining a heterogeneous computing environment, more companies are striving to use Windows NT as their Operating System (OS) at all levels of the industrial hierarchy. Its use as a network server system or as a desktop system is easy to understand, since these are the very environments for which Windows NT was designed. However, there is also an impetus to use it in other environments, such as the factory floor. A common characteristic of these environments is that they often require hard real-time system behavior.

Can Windows NT fulfill this need? The answer is, not as delivered. However, with additional software, it is possible to realize hard real-time performance on Windows NT. The remainder of this paper justifies these statements and describes VenturCom's RTX environment including RTSS, a Real-Time SubSystem, for Windows NT running on the PC architecture (i.e., Intel x86 and compatible systems).

An earlier paper [Carpenter 97] discussed this effort during its development. This paper offers a closer look at the actual implementation as delivered, including performance numbers and enhancements made in the meantime, as well as a look at future areas for development.

2 Windows NT and the Real-time World

2.1 What it means for a system to be real-time

A real-time system is one in which the correct operation of the system depends not only on the results that are delivered, but **when** they are delivered. It is important to note that "real-time" does not necessarily mean "fast"; rather it refers to how deterministic the response time characteristics of the system are. That is, the important measure is not average response time but worst-case response time. Real-time systems are sometimes further classified as *hard* or *soft* real-time systems. A hard real-time system is one in which the response time determinism requirement is absolute; for a soft real-time system, some small deviations are tolerated. A fundamentalist viewpoint would consider "soft real-time" to be an oxymoron, and for the remainder of this paper, when we say "real-time" we mean hard real-time.

An example of a real-time system is a system controlling a piece of capping machinery over a conveyer belt transporting bottles to be capped. It is not enough for

the system to correctly position the cap dispenser; it must do so when a bottle is in position to be capped. All the accuracy in positioning is worthless if the dispenser arrives in position after a bottle has passed.

In addition to this determinism, there are a number of other requirements that real-time systems have typically come to provide:

- A multithreaded, preemptive scheduler with a large number (typically 64-256) of thread priorities.
- Predictable thread synchronization mechanisms.
- A system of priority inheritance.
- Fast clocks and timers.

2.2 Why is stock Windows NT unsuitable as a real-time system?

Microsoft Windows NT has been designed as a general-purpose operating system, suitable for use both as an interactive system on the desktop and as a server system on a network [Solomon 98]. The shortcomings of NT in real-time applications have been thoroughly researched [Ramamritham 98] [Timmerman & Monfret 96]:

- Too few thread priorities.
- Opaque and non-deterministic scheduling decisions.
- Priority inversion, particularly in interrupt processing.

The logic of RTX design is dictated by several factors. The stock NT operating system is a mass-market product, not readily tweaked for niche applications like real-time. While Microsoft-sponsored research into real-time NT has produced some interesting results [Sommer 96], especially for cases when applications advertise their resource requirements in advance [Sommer 97][Sommer & Potter 96], it is doubtful that this operating system, aiming for a very broad market, should absorb the overhead and complexity of real-time functionality [Microsoft 95]. This factor suggests that the proper way to add real-time to NT is via an extension, or a plug-in to the generic product [Jones 98].

2.3 Why extend Windows NT to a real-time system?

At the same time, NT offers a very rich and sophisticated device driver model. That, and the customizable Hardware Abstraction Layer (HAL), offer a developer great flexibility and control over system behavior, and scope for creativity in tackling technical challenges. Thus, real-time functionality can be implemented “by the book” following the Microsoft NT Device Driver Kit (DDK) and HAL models [Baker 97].

Finally, for a writer of NT kernel extensions who is not employed by Microsoft, the NT kernel is akin to silicon, as its interfaces and behavior are fixed. Rather than lament the fact, one can make virtue of necessity and produce a compact design for the extension, easily portable between different versions of NT, and, indeed, between NT and other operating systems such Windows CE or Unix. Below we illustrate how RTX lived up to these portability goals.

3 So, You Want to Write A Hard Real-Time NT Environment?

3.1 Why extend Windows NT to a real-time system?

Given that many of the shortcomings of NT just mentioned are due to its thread model and thread scheduler, it is logical that the extension have its own thread model with its own scheduler. Likewise, the NT syn-

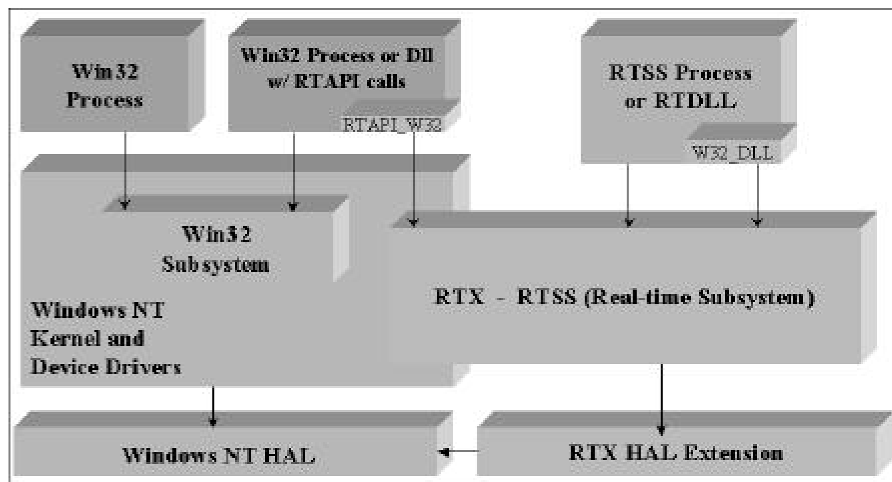


Figure 1. RTX Architecture

chronization objects such as events, semaphores, and mutexes lack the necessary real-time semantics (in particular, they neither ready threads waiting on an object in priority order nor prevent priority inversion). For these reasons, the extension should implement its own synchronization objects [Bollella 95].

If, following the logic of the NT environment, you have decided to implement a hard real-time subsystem for NT, your real-time environment should be able to:

- Preempt NT anywhere, at least outside critical NT interrupt-processing code.
- Defer NT interrupts and faults while running real-time tasks.
- Process real-time interrupts while running real-time tasks.

The notion of preempting high-level IRQ activity of NT and its drivers for unbounded periods of real-time activity may strike one as dangerous. Yet, such events are commonplace, and NT is designed to handle them: high-IRQL events intrude on lower-IRQL ones, bus-mastering by DMA peripherals may defer even the highest-level interrupt processing, and PCI devices may stall CPU accesses to the I/O space (see RTX and Interrupt Latency). Thus, from the NT point of view, RTSS activity that steals its cycles is equivalent to taking and coming back from an interrupt. Such an event is well handled by NT, regardless of its duration.

Functional needs of the real-time subsystem would include IPC with the Win32 subsystem, access to the NT kernel functionality (interrupt management, port I/O, shutdown/crash handlers), and – importantly – compatibility with Win32, at least at the source code level.

3.2 RTX Structure

RTX is implemented as a collection of libraries (both static and dynamic), a real-time subsystem (RTSS) realized as an NT kernel device driver, and an extended HAL (see Figure 1) [Carpenter 97]. The subsystem implements the real-time objects and scheduler previously mentioned. The libraries provide access to the subsystem via a real-time API (RTX API). RTX API provides access to these objects. Note that the RTX API is callable from the standard Win32 environment as well as from within RTSS. While using RTX API from Win32 does not provide the determinism available within RTSS, it does allow much of the application development to be done in the friendlier Win32 programming environment rather than that provided by the DDK [Anschuetz 98]. All that is necessary to convert a Win32 program to an RTSS program is to relink with a different set of libraries.

RTX applications – just as RTX itself – are implemented on top of loadable NT drivers, although they lack the I/O Manager-related hooks. It is a natural fit: Windows NT drivers are process-like from the NT Service Manager's point of view, controllable by users, and get loaded into the kernel address space.

4 RTX in Depth

4.1 The Real-time HAL

The HAL is the one piece of the Windows NT system whose source is available for modification and extension. RTX has modified the HAL for three purposes:

1) To add interrupt isolation between NT and RTSS threads.

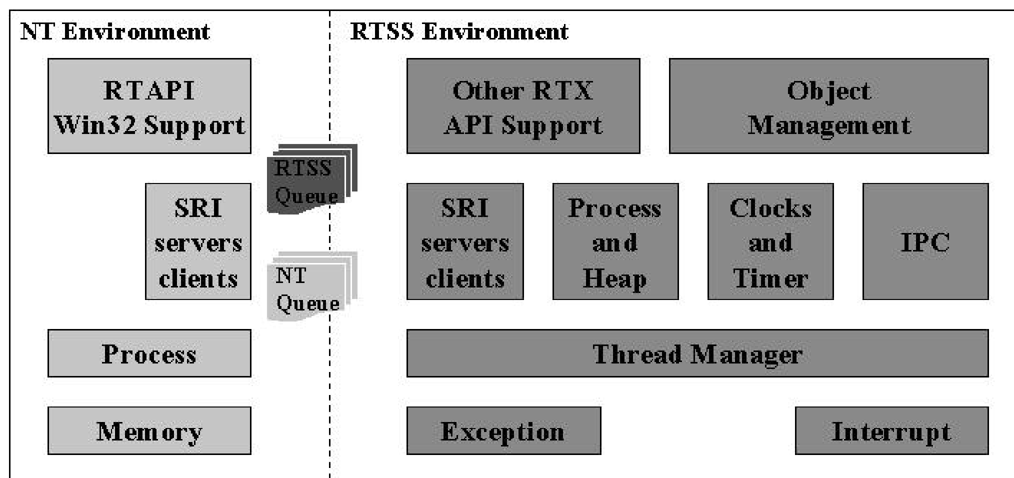


Figure 2. RTSS Detailed Architecture

2) To implement high-speed clocks and timers.

3) To implement shutdown handlers.

Interrupt isolation means that it is impossible for an NT thread or an NT-managed device to interrupt RTSS. It is also impossible for an NT thread to mask an RTSS-managed device. The HAL ensures that these conditions are met by controlling the processor's interrupt mask. When running an RTSS thread, all NT-controlled interrupts are masked out. When an NT thread calls to set the interrupt mask, the HAL, which is the software that actually manipulates the mask, ensures that no RTSS-controlled interrupt is masked out.

NT provides clocks and timers with a maximum resolution (i.e., smallest granularity) of 1 msec. The RT-HAL extends this to 1 μsec. It also provides access to the second clock on the PIC.

4.1.1 Protection from NT Crashes

In addition to interrupt management and fast-timer services, the real-time HAL also provides NT shutdown management. An RTSS application can attach an NT shutdown handler for cases where NT performs an orderly shutdown, or crashes, producing a so-called Blue (Stop) screen. An orderly shutdown allows RTSS to continue unimpaired and resumes when all RTSS shutdown handlers return. In a blue-screen shutdown, RTSS shutdown handlers run with certain limitations, unable to call NT services (e.g., for memory allocation) or to handle faults. In practice, it means that a shutdown handler should clean up and reset any hardware state, possibly alert an operator, or switch to a hot spare when the system stops, due to either a normal shutdown or a crash.

4.2 Extending the HAL

Whereas earlier releases of RTX replaced the HAL, the latest one – RTX 4.3 – extends the standard NT HAL dynamically. The HAL extension driver, starting at OS initialization time (SERVICE_SYSTEM_START), performs dynamic HAL detection in memory, intercepts interrupt, timer, and shutdown-related calls, and re-directs them to their RTX counterparts. This binary hooking technique has a number of advantages over HAL replacement:

- RTX handles a broader range of OEM platforms, as call re-direction is limited to calls which vary little among different OEMs and SPs.
- RTX is compatible with a greater range of NT Service Pack (SP) releases, as we do not need to merge RTX changes with the latest SP's HAL sources.
- Installation becomes more robust, as the on-disk copy of the HAL is untouched, hence RTX is unaffected by SP upgrades.
- Upgrades to newer versions of NT become easier, if not effort-free.

The benefits of the HAL extension were demonstrated when RTX has installed and run successfully on Windows 2000 (neè NT 5.0) Beta 2. This required no development effort on Win2000, although effort certainly will be needed to improve performance to NT4.0 levels.

4.3 RTX and Interrupt Latency

4.3.1 Software Causes of Latency

A switch from NT to RTSS happens on an interrupt, either from the RTX high-speed clock, or from another device generating RTX interrupts. Therefore, achieving RTX ISR determinism requires reducing NT interrupt latency. Let us examine the sources of this latency.

The most significant is IRQ masking by the NT kernel and drivers, routinely done for periods up to several milliseconds via NT *KeRaise/LowerIrql* calls. The NT kernel, HAL, and certain special drivers also perform processor-level masking of all interrupts via x86 STI/CLI instructions, for periods of up to 50 μsec.

NT and RTX interrupt processing, naturally, masks interrupts, thereby adding to ISR latency. Although NT relies very heavily on interrupts in many situations (e.g., raising software exceptions, or unwinding a thread's stack), the NT interrupt sequence is reasonably compact in its contribution to the worst-case ISR latency.

4.3.2 Hardware Causes of ISR Latency

The most obvious hardware-related problem is cache dirtying and flushing by applications and the operating system. This category also includes re-filling of the TLBs. Video drivers are particularly aggressive users of caches, causing contention-related flushes when an RTX interrupts starts running. Histograms of ISR behavior in the presence of cache-dirtying applications would typically have a double-hump profile, with most samples near the best-case band, and another large number of samples in the flushing-related band (see Figure 3).

Power management, especially on portable devices, creates occasional long-latency events when the CPU is put in a low-power-consumption state after a period of

inactivity. Such problems are usually quite easy to detect. A typical system can disable those features via BIOS setup.

Bus mastering events can cause long-latency CPU stalls. Such cases include high-performance DMA SCSI devices, causing CPU stalls for periods of many microseconds, or video cards that insert wait cycles on the bus in response to a CPU access. Sometimes the behavior of such peripherals can be controlled from the driver, trading off throughput for lower latency.

While no operating system can protect an application against such hardware factors, RTX offers a panoply of tools to diagnose platform-related latencies, and identify the misbehaving peripherals. Being mindful of such factors and using RTX tools to qualify one's development platform are essential for a system's overall performance.

4.4 RTX Interrupt Latency Reduction Techniques

RTSS entirely eliminates latencies from IRQL masking by NT and NT drivers. The RT HAL performs interrupt isolation, re-programming the PIC when switching between NT and RTSS. The result is that RTX interrupts can always interrupt NT, while RTX masks all NT interrupts while RTSS is running.

Processor-level interrupt masking, on the other hand, can not be defeated, other than through the perilous use of x86 NMIs (non-maskable interrupts). RTX adopts a static solution hooking gratuitous cases of interrupt preemption (e.g., page-zeroing operations) to use IRQ locks instead. The RTX Dynamic Hook driver scans the NT kernel for signatures of such operations, hooking them to use spin locks (or IRQ-based synchronization on a uniprocessor) instead.

These techniques provide worst-case interrupt latencies of under 50 microseconds on a typical 200MHz PC platform.

4.5 RTX Objects

The RTSS Environment has a fast streamlined object manager (see Figure 2). The objects it supports satisfy the following criteria: 1) Usefulness for real-time programming, and 2) Compatibility with Win32. The IPC objects are also available to Win32 applications and device drivers, allowing programmers to harness the full power of NT. The IPC set includes mutexes, events, semaphores, and shared memory objects.

The RTSS object manager uses the Windows NT non-paged memory pool for its storage requirements. There are advantages and disadvantages to this approach. Using kernel-provided mechanisms decreases RTX's development time and resource consumption. Object allocation, however, is non-deterministic.

4.6 RTSS Scheduler

The RTSS scheduler implements a priority based preemptive policy with priority promotion to prevent priority inversion. The RTSS environment provides for 128 priority levels, numbered from 0 to 127, with 0 the lowest priority. The RTSS scheduler will always run the highest priority thread that is ready to run (in the case of multiple ready threads with the same priority, the thread which has been ready the longest will run first). An RTSS thread will run until a higher priority ready thread preempts it or until it voluntarily relinquishes the processor by waiting (there is no time-slicing among ready threads at the same priority).

The scheduler has been coded with the requirements of real-time processing in mind. Most importantly, its operation is low latency, and is unaffected by the number of threads it is managing. Each priority has its own ready queue, maintained as a doubly linked list. This allows the execution time of insertion (at the end of the list) and removal (from anywhere in the list) to be independent of the number of threads on the list. A bit array keeps track of which lists are non-empty, and manipulating this bit array is done by high-speed assembly-coded routines.

While an RTSS thread is running, all NT-managed interrupts, as well as any interrupts managed by threads of a lower priority than the current thread, are masked out. Conversely, all interrupts managed by higher priority threads are unmasked, allowing for a higher-priority thread to preempt the current thread. In addition to these device interrupts, other mechanisms that can cause the currently running thread to be preempted are the expiration of a timer that causes a higher priority thread to become ready, or the signaling of a synchronization object (by the currently running thread) for which a higher priority thread is waiting.

In order to deal with priority inversion, RTSS implements the classic solution [Nakajima 93] [Sha 90], *priority promotion,* to prevent this situation. For the duration of the time that a low priority thread owns an object for which a high priority thread is waiting, its effective priority is promoted to that of the high priority thread.

4.7 Service Request Interrupt (SRI)

An important architectural feature of RTX is its lock-less interrupt-driven interface between NT and RTSS. This clean architectural separation has enabled ports of RTSS to various environments (e.g., multiprocessor RTX product, and RTSS demo for Windows CE2.0), while ensuring a fast and robust implementation. The NT side of the RTX driver and the RTSS environment communicate by inserting commands into one of the two buffer queues (one in each direction) and initiating a Service Request Interrupt (SRI) to request service by the other side. A server thread executes a request and a reply message is posted in the other buffer. A typical NT-to-RTSS request is an IPC operation like WaitForSingleObject or a Release operation on a RTSS object. A typical RTSS-to-NT operation is a memory allocation or a file I/O request. The SRI design favors lower response time over throughput, responding to an RTSS request as soon as possible.

4.8 Win32-RTSS IPC

Inter-environment IPC is a key feature of RTX, allowing tightly integrated applications where hard real-time processes run in the more resource-intensive RTSS environment, and the rest of the application runs in the Win32 subsystem. This section describes the IPC design.

4.8.1 RTSS Proxy Model

IPC, as the rest of the NT-RTSS communication, uses the SRI channel. Given that the SRI channel prevents NT threads from queuing directly for RTX objects, RTX uses *proxy* processes and threads to support blocking IPC from Win32. When a Win32 thread accesses an RTX object, RTSS uses a proxy thread on its behalf. This model is clean and economical, its advantages being:

- No state-keeping on the NT side for blocking IPC requests.
- No special-casing in RTSS for external Win32 wait requests.
- Handle and object cleanup for Win32 process and thread termination is handled automatically by RTSS proxy process/thread cleanup.

Although proxies involve some memory and CPU overhead, the clean design and quick implementation were worth the tradeoff.

4.8.2 Taming the NT I/O Manager

Preserving seamless Win32 semantics and achieving good performance for cross-environment IPC presents several challenges.

The NT 4.0 DDK provides no exposed interfaces for driver thread notification in case a Win32 thread using that driver terminates. Yet, Win32 mutex semantics, require such a mechanism. An RTX mutex acquired, but not released, by a thread at the time of the thread's termination, must be marked as "abandoned", indicating that the shared data it protects may be inconsistent. To implement thread-termination cleanup, RTX takes advantage of I/O Manager's IRP (I/O Request Packet) cleanup: each thread attached to the RTX Win32 Dynamically Linked Library (DLL) sends a "death IRP" to the RTX driver. When the thread terminates, NT calls this IRP's cancel routine, notifying the RTX driver and, thereby, the RTSS object layer. The I/O Manager presents a powerful, yet often challenging environment, as its event delivery is asynchronous. E.g., calls to the MJ_CLEANUP driver dispatch routine and the cancel routine call can come in any order, requiring careful synchronization for the RTX driver's per-thread and per-process structures.

Performance of the Win32-RTSS IPC presented another concern. In an early implementation, the total latency of an uncontested RtWaitForSingleObject call from Win32 averaged 130µsec. Analysis has shown that about 40µsec of the total (over 30%) was spent in the NT I/O Manager. Therefore, RTX4.2 has undergone a redesign of the IPC code, using direct signaling and shared memory between the RTX driver and the Win32 IPC client [Tomlinson 97]. RTX user and kernel threads share synchronization objects, signaling each other directly, thus shrinking the overhead and the latency of the NT I/O manager by a factor of four.

Note that operations on RTX synchronization objects locked by Win32 applications become non-deterministic [Carpenter 97]: as any RTSS thread can preempt an NT thread holding such a lock, causing an apparently unbounded case of priority inversion. This, however, is a matter of application design: locking an object shared from Win32 should be left to a non-critical RTSS thread. Furthermore, this issue is ameliorated when RTSS and NT run on different dedicated processors, in a multiprocessor RTX system.

4.9 Fast Timer Support

On all PC platforms real-time HAL provides clock resolution of 1µsec or better, and timer period of

100μsec or better. When RTSS is not used, there are no timing differences between a real-time HAL and a regular HAL systems.

4.10 Dynamically Linked Libraries

No tour of Win32 would be complete without a mention of DLLs. RTSS supports Win32 DLL API (LoadLibrary, GetProcAddress). Currently, all the static and global data in an RTSS DLL is shared between all RTSS processes attached to that DLL.

4.11 Structured Exception Handling in RTSS

Structured Exception Handling (SEH) is a relatively little known but rather important feature of Win32 and NT kernel environments. Its pedigree goes back to OS/2 and OSF Unix implementation. SEH provides exception handling via *try/except* and *try/finally* constructs of the Microsoft C implementation. C++ exception handling is layered on top of SEH, as are libc *signal/raise* calls, making SEH a necessity for any Win32-compliant environment. The salient features of this model are:

- Compiler-specific exception handlers.
- Operating system-specific stack unwinder and exception dispatcher routines.
- User-supplied exception filters.
- Two-stage exception handling algorithm which first invokes the OS-specific dispatcher routine to scan the thread's stack backwards calling filters in search of a suitable handler, then the OS-specific unwinder to roll back the stack, if necessary.
- Last-chance default and user-supplied exception routines.
- A special mechanism for nested exceptions and "collided" unwinds.

The RTSS SEH implementation maintains compatibility with Microsoft structures, handler-calling conventions, SEH API behavior, etc. In addition, it is engineered for real-time, to minimize processor-level interrupt masking and disruptions to RTSS threads:

- Win32 generates a software interrupt when raising a software exception via the Win32 RaiseException API; RTSS calls a user-mode exception dispatcher.
- Win32 SEH uses a special interrupt when it sets a new user context after unwinding the stack; RTSS restores context in user mode.
- NT may "edit" (move) an exception trap frame with interrupts disabled; RTSS does this in user mode.

For hardware exceptions, the RTSS algorithm doctors the trap frame to call the RTSS exception dispatcher; then "returns" from the ISR to the dispatcher, and handles the exception. Thus, a software exception involves no ISR latency penalty for other threads; a hardware exception only adds the worst-case penalty of a single interrupt.

5 Performance

Table 1 presents selected performance numbers for a recent release of RTX.

Table 1. Typical latencies for a 266MHz Pentium II

Operation	*Latency (mec)*
Interrupt thread: avg., max	9, 38
Timer interrupt: avg., max	10, 43
Context switch (yield): avg.	3.5
Context switch (semaphore): avg.	5.5
Thread priority change: avg.	3
Thread yield: avg.	3
Acquire semaphore, uncontested: avg.	1.5
Acquire semaphore: avg.	5
Win32-to-RTSS semaphore call: avg.	50

Figure 3 is a typical look from the RTSS interrupt latency measurement utility in the presence of a typical NT workload: disk searches, video updates, network activity, screen saver, etc. The lower and narrower sections of the chart's bars represent activity during the last second only.

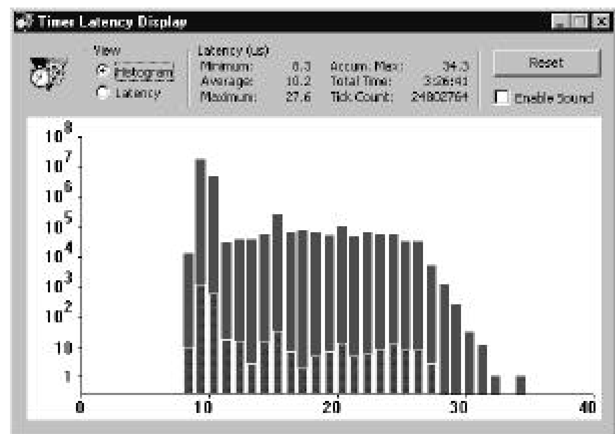


Figure 3. RTSS timer interrupt latency histogram for a typical workload. X axis – μsec, Y axis – number of samples.

Figure 4 is the Win32 counterpart for the same kind of workload on the same machine – Gateway GP5-166 running NT4.0 Workstation SP4. An astute reader may point out that the first histogram describes Win32 threads, whereas the second – kernel driver threads. However, the difference between the two amounts to the several microseconds of kernel-user transition overhead for the Win32 threads, so the worst-case latencies of NT driver and Win32 threads differ very little under the same workload.

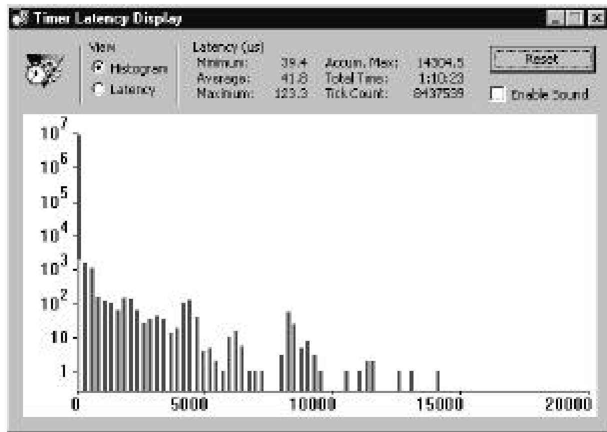


Figure 4. Win32 timer interrupt latency histogram for a typical workload. X axis – µsec, Y axis – number of samples.

Copious performance results are available from independent evaluations of RTX and other real-time NT extensions [OMAC 98] [Real-Time 98] [Timmerman 98].

5.1 Performance Tools

RTX performance tools enable developers to qualify and tune performance on their platforms. *Ksrtm* is a driver and a Win32 utility for HAL-level timer interrupt latency measurement. Running in the kernel makes it relatively insensitive to cache jitter. *Ksrtm* also determines which NT component or device driver is causing the greatest latency event: after such an event, the *Ksrtm* ISR obtains the address of the interrupted sequence from the stack, then resolves it to a loaded NT kernel module. The *Srtm* application is a simple RTX API timer latency measurement tool running either in RTSS or Win32, which produces a histogram, realistically reflecting timer latency observed by an application. The *Lpt* tool determines long-latency events due to bus mastering events.

6 Future Directions

6.1 Multiprocessing

The initial releases of RTX ran on single processor systems. The most recent release runs on multiprocessor systems that conform to the Intel Multiprocessor Specification Rev 1.4.

This specification provides for interrupts to be controlled by an Advanced Programmable Interrupt Controller (APIC), suitable for a multiprocessor system. Through the APIC, different interrupts can be steered to different sets of processors. RTSS dedicates one processor of the system to running RTSS threads, while the remaining processors run NT threads. This dramatically lessens the latency of real-time threads (from 50 µsec to less than 13 µsec,) while preventing the processor starvation of NT threads, possible on a single processor system

6.2 Ways to Grow

RTSS can serve as a basis for other real-time environments, like Newmonics' real-time RTSS-based Java-compatible virtual machine [Nilsen 98] [Nilsen & Lee 98]. It also can be linked with other NT subsystems such as Interix by Softway [Walli 97]. Other possibilities include a COM interface to RTSS objects accessed from Win32 or, indeed, making COM available within the RTSS environment.

7 Conclusion

VenturCom's RTX has shown that, with a selected collection of extensions, it is possible to augment Windows NT to provide the features of a real-time operating system at the same time it continues to be used as a general purpose platform. The resulting system meets the constraints of determinism which are a necessary part of the real-time world, while providing an environment more familiar to a wide body of users.

Availability

RTX is available from www.vci.com.

References

[Anschuetz 98] E. Anschuetz, M. Biddle, S. Giambarberee, B. Riner, J. Dube, *Real Time Flight Simulators Under NT*, Proceedings of the 1998 Interservice/Industry Training, Simulation and Education Conference (I/ITSEC 12 1998).

[Baker 97] Art Baker, *The Windows NT Device Driver Book*, Prentice Hall, 1997.

[Bollella 95] G. Bollella and K. Jeffay, *Support For Real-Time Computing Within General Purpose Operating Systems: Supporting co-resident operating systems*, Proc. IEEE Real-Time Technology and Applications Symposium Chicago, IL, May 1995.

[Carpenter 97] Bill Carpenter, Mark Roman, Nick Vasilatos, Myron Zimmerman*, The RTX Real-Time Subsystem for Windows NT*, Usenix Windows NT Symposium, 1997.

[Jones 98] C. Jones, M. Cherepov, *Windows-based systems and the Win32 API*, Real-Time Magazine 98/3.

[Microsoft 95] Microsoft, *Real-Time Systems and Microsoft Windows NT;* http://www.msdn.microsoft.com.

[Nakajima 93] T.Nakajima, T.Kitayama, H.Arakawa, H.Tokuda , *Integrated Management of Priority Inversion in Real-Time Mach*, IEEE Real-Time Systems Symposium, December 1993.

[Nilsen 98] Kelvin Nilsen, Simanta Mitra, Sairam Sankaranarayananan, Venkatesh Thanuvan, *Asynchronous Java Exception Handling in a Real-Time Context*, Workshop on Programming Languages for Real-Time Industrial Applications '98, Madrid, Spain, December 1, 1998.

[Nilsen & Lee 98] Kelvin Nilsen, Steve Lee, *Perk(TM) Real-Time API,* July 1998, Newmonics, Inc. www.newmonics.com/WebRoot/perc.info/perc.api.pdf.

[OMAC 98] Open Modular Architecture Controls (OMAC) Users Group, *Hard Real-time Extensions of Windows NT© Evaluation Report Test Plan and Phase 1 & 2*; www.arcweb.com/omac/Docs&NRs/ntrtrpt2.pdf.

[Ramamritham 98] Krithi Ramamritham, Chia Shen, Oscar Gonzalez, Shubo Sen, Shreedhar B Shirgurkar, *Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations* Proceedings of IEEE RTAS'98 (the IEEE Real-Time Technology and Applications Symposium), June 3-5, 1998. Denver, Colorado.

[Real-Time 98] Real-Time Magazine, *Windows NT RT Extensions - Evaluation Reports;* www.realtime-info.be/encyc/market/rtos/eval_roadmap.htm.

[Sha 90] L. Sha, R. Rajkumar, and J. P. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, IEEE Transactions on Computers, 39 (9):1175-1185, Sept. 1990.

[Solomon 98] David A. Solomon, *Inside Windows NT*, *Second Edition*, Microsoft Press, 1988.

[Sommer 96] Sommer, S., *Removing Priority Inversion from an Operating System,* Proceedings of the Nineteenth Australasian Computer Science Conference (ACSC'96), Melbourne, Australia, January 31 - February 2, 1996.

[Sommer & Potter 96] S. Sommer and J Potter, *An Overview of the Real-Time Dreams Extensions* Proceedings of the Third Australasian Conference on Parallel and Real-Time Systems, September 1996.

[Sommer 97] Sommer, S., *Dreams in a Nutshell*, Proceedings of the USENIX Windows NT Workshop, Seattle, Washington, August 11-13, 1997.

[Timmerman & Monfret 96] M. Timmerman and J. Monfret, *Windows NT as Real-Time OS?* Real-Time Magazine; http://www.realtime-info.be.

[Timmerman 98] Martin Timmerman, Bart Van Beneden, Laurent Uhres, *Windows NT Real-Time Extensions: better or worse?* Real-Time Magazine - 98/3.

[Tomlinson 97] Paula Tomlinson, *Understanding NT: Signaling Apps from Drivers,* Windows Developer's Journal, March 1997.

[Walli 97] Stephen R. Walli, *OPENNT: UNIX Application Portability to Windows NT via an Alternative Environment Subsystem,* Proceedings of the USENIX Windows NT Workshop, Seattle, Washington, August 11-13, 1997.