IntervalZero

Hard Real-Time with IntervalZero RTX® on the Windows® Platform

Abstract

Microsoft® Windows® is increasingly the platform of choice for deployment of embedded systems. To meet the stringent performance criteria for embedded systems that require hard real-time within the Windows environment, developers augment Windows' capabilities with IntervalZero's RTX software.

This White Paper examines the complementary aspects of Windows and IntervalZero RTX in embedded real-time systems. RTX provides a Real-Time Subsystem (RTSS) that runs on Windows and is also capable of performing symmetric multiprocessing (SMP) on systems with more than two processors.

RTX implements deterministic scheduling of real-time threads; provides inter-process communication mechanisms between the real-time environment and the native Windows environment; and extends Windows to enable capabilities that are generally found only in proprietary real-time operating systems.

This paper takes an in depth look at how RTX provides these capabilities.

Introduction

Windows' embedded market share and popularity continue to increase for a variety of reasons:

- Windows is consistently the most productive embedded development environment. Studies have shown that Windows embedded project development is 43% faster and costs 68% less on average than projects developed on Linux.
- Windows longevity and stability, as well as its increasing market share, position it as the safest choice.
- Windows ubiquitous interface lowers end-user training costs.

Importantly, Windows is already in use for the human-machine interface (HMI) in many embedded systems and there is an additional back-end system providing the required real-time functionality.

By adding RTX, a single system can deliver both a world-class user interface and real-time capability. The added complexity and cost of maintaining heterogeneous computing environments continues to push more companies to standardize on Windows as their operating system (OS) at all levels of an organization – including environments that very often require hard real-time system behavior such as on the factory floor, in medical devices, and in simulation, test and communications systems.

While it is true that Windows cannot deliver hard real-time right out of the box – it can with IntervalZero's RTX.

This paper looks closely at implementation of RTX with Windows.

Windows and the Real-time World

What it means for a system to be real-time

A real-time system depends not only on **what** results are delivered, but also **when** those results 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. 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, meaning that if the information is received after the deadline it is no longer valid.

For a soft real-time system, some small deviations are tolerated. Information becomes less relevant but still useful. In this paper, "real-time" means hard real-time.

An example of a real-time system would be a system that controls bottle-capping machinery. It is not enough for the system to correctly position the cap dispenser; it must do so at the precise moment 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 by on the conveyor belt.

In addition to determinism, there are a number of other requirements that real-time systems typically require:

- A multi-threaded, preemptive scheduler with a large number (typically 64-256) of thread priorities
- Predictable thread synchronization mechanisms
- A system of priority inheritance
- More precise clocks and timers

Windows and Real-time

Windows is general-purpose operating system, suitable for use both as an interactive system on the desktop and as a server system on a network. The shortcomings of Windows in real-time applications include:

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

Windows provides timers with a maximum resolution (i.e., smallest granularity) of 1000 μs (1 millisecond). RTX lowers this to 1 μs where supported by the hardware.

It is worth pointing out that Microsoft has been very clear that its Windows roadmap **does not** call for adding real-time functionality.

Extending Windows to Deliver Hard Real-Time

Given that many of the Windows real-time shortcomings mentioned above are related to its thread model and thread scheduler, it is logical that any real-time extension would have its own thread model with its own scheduler.

Additionally, because the Windows synchronization objects – such as events, semaphores, and mutexes – lack the necessary real-time capabilities (in particular, they do not ready threads waiting on an object in priority order, nor do they prevent priority inversion) a Windows real-time extension should implement its own synchronization objects.

If you have decided to implement a hard real-time subsystem for Windows, your real-time environment should be able to:

- Preempt Windows any time, at least outside critical Windows interrupt-processing code

- Defer Windows interrupts and faults while running real-time tasks
- Process real-time interrupts while running real-time tasks

Although preempting the high-level interrupt request (IRQ) activity of Windows and its drivers for unbounded periods of real-time activity may strike some as dangerous, such events are commonplace, and Windows is designed to handle them. For example:

- High-IRQL events intrude on lower-IRQL ones
- Bus-mastering by DMA peripherals and System Management Mode processing defer even the highest-level interrupt processing, and PCI devices may stall CPU accesses to the I/O space

RTX Structure

RTX is implemented as a collection of libraries (both static and dynamic), a real-time subsystem (RTSS) realized as a Windows kernel device driver, and an extension to the Windows HAL.

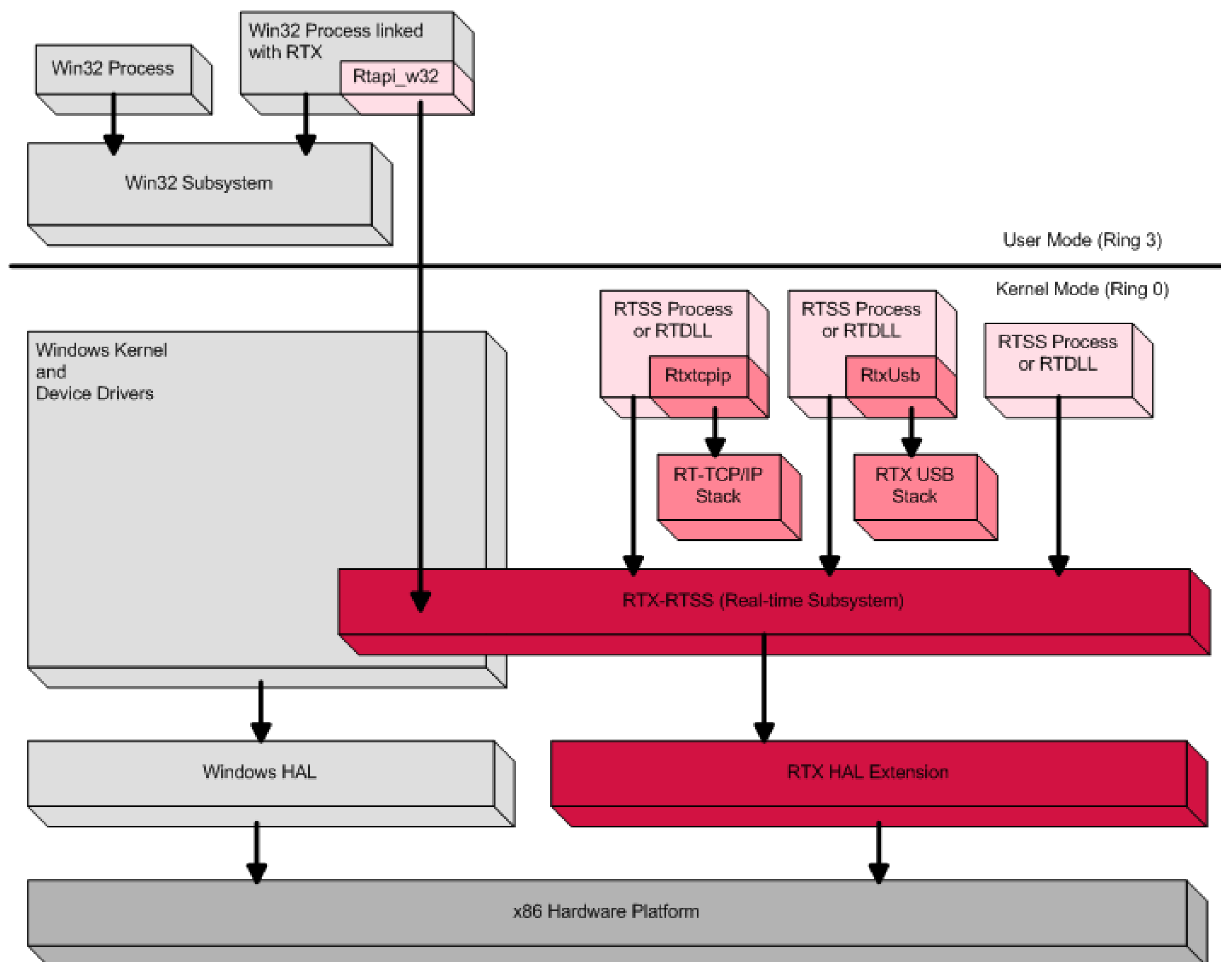


Figure 1. RTX Architecture

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

The subsystem implements the real-time objects and scheduler. The libraries provide access to the subsystem via a real-time API (RTAPI). Note that some of the RTAPI functions are callable from the standard Win32 environment as well as from within RTSS.

Calling RTAPI from Win32 does not provide the determinism available within RTSS, but it does allow non real-time and real-time applications to communicate and pass information back and forth. All that is necessary to convert a Win32 program to an RTSS program is to relink with a different set of libraries.

RTX in Depth

System Configurations

Every embedded system is different and has a different set of requirements. RTX provides the flexibility to configure the combined Windows/RTX system to best optimize the system for the intended purpose.

For example, one embedded system may only have a single, light-weight real-time task. This type of embedded functionality would not require a large amount of processing time from the CPU and could easily share the CPU with Windows. By contrast, another embedded system may have a larger number of real-time or critical tasks that all have different timing requirements. In this case, it would be best to be able to have these different tasks executing in multiple threads, and if available, on multiple processors.

RTX gives system designers and developers the tools to be able to accomplish the above scenarios – and anything in between. RTX provides an environment that can run on a single processor with Windows or take advantage of multiple processors via Symmetric Multiprocessing (SMP) that is optimized for embedded systems.

Non-Multiprocessing Configurations

Non-Multiprocessing is used to describe the use of RTX on a single, or multiple processor system, with RTX executing only on a single processor.

There are two modes that RTX can run in under this scenario:

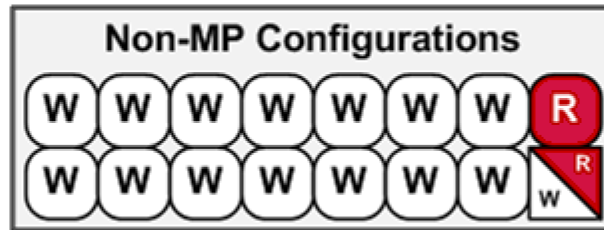
- **Shared Mode**

  In shared mode, RTX would execute on a processor that Windows also executes threads on. The light-weight real-time task scenario above would be a good example of when you would use this configuration. The system has a very simple real-time requirement, allowing Windows to have all the additional processing power of the system.

- **Dedicated Mode**

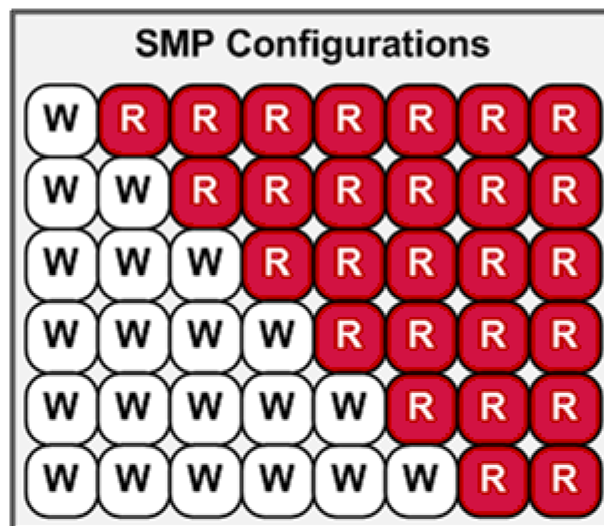
  In dedicated mode, RTX exclusively executes its threads on a single core. This mode requires at least a two-processor system, which can be in the form of distinct physical processors or multi-core. Dedicated Mode would be used instead of Shared Mode when more significant real-time processing needs to occur.

The graphic below shows how these two Non-multiprocessing configurations would look logically. The "W" represents a processor where Windows threads execute. The "R"s represent processors where the real-time threads execute.



Symmetric Multiprocessing Configurations

SMP configurations require a system with multiple processors/cores that can assign 2 to 7 processors/cores exclusively to the real-time subsystem. This is demonstrated by the graphic below. It is worth noting that an RTX SMP Runtime also supports non-MP configurations.



RTX with SMP provides for seamless execution of real-time threads across processors via a single RTSS instance. This is important in that there is little to no resource-usage redundancy as would be the case with an asymmetric multiprocessing (AMP/ASMP) design.

The RTX SMP implementation is optimized for embedded systems. RTX provides mechanisms to aid the embedded developer in creating a system that meets the design requirements, including:

- Explicit thread migration
- Deterministic thread/process level processor affinity
- IRQ/ISR/IST level processor affinity for deterministic device I/O
- Accurate thread timing API for effective load balancing

The Real-time Hardware Abstraction Layer (HAL) Extension

The RTX HAL Extension module performs dynamic HAL detection in calls related to memory – interrupts, timer, shutdown – 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, call re-direction is limited to calls that vary little among different OEMs and Service Providers
- Installation becomes more robust, as the on-disk copy of the HAL is untouched, RTX is generally unaffected by Windows Hot Fixes and Service Packs

The Windows HAL is extended by RTX for three purposes:

- To add interrupt isolation between Windows and RTSS threads
- To implement high-speed clocks and timers
- To implement shutdown handlers

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

As noted earlier, Windows provides timers with a maximum resolution (i.e., smallest granularity) of 1000 μs (1 millisecond). The RTX Hal Extension lowers this to 1 μs, where supported by the hardware.

Protection from Windows STOP Messages

In addition to interrupt management and fast-timer services, the RTX HAL Extension also provides Windows shutdown management. An RTSS application can attach a shutdown handler for cases where Windows performs an orderly shutdown, or halts with a bug check, commonly called a Blue Screen.

An orderly shutdown allows RTSS to continue unimpaired and resumes when all RTSS shutdown handlers return. In a bug check case, RTSS shutdown handlers run with certain limitations, unable to call Windows services (i.e., new memory allocation). 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.

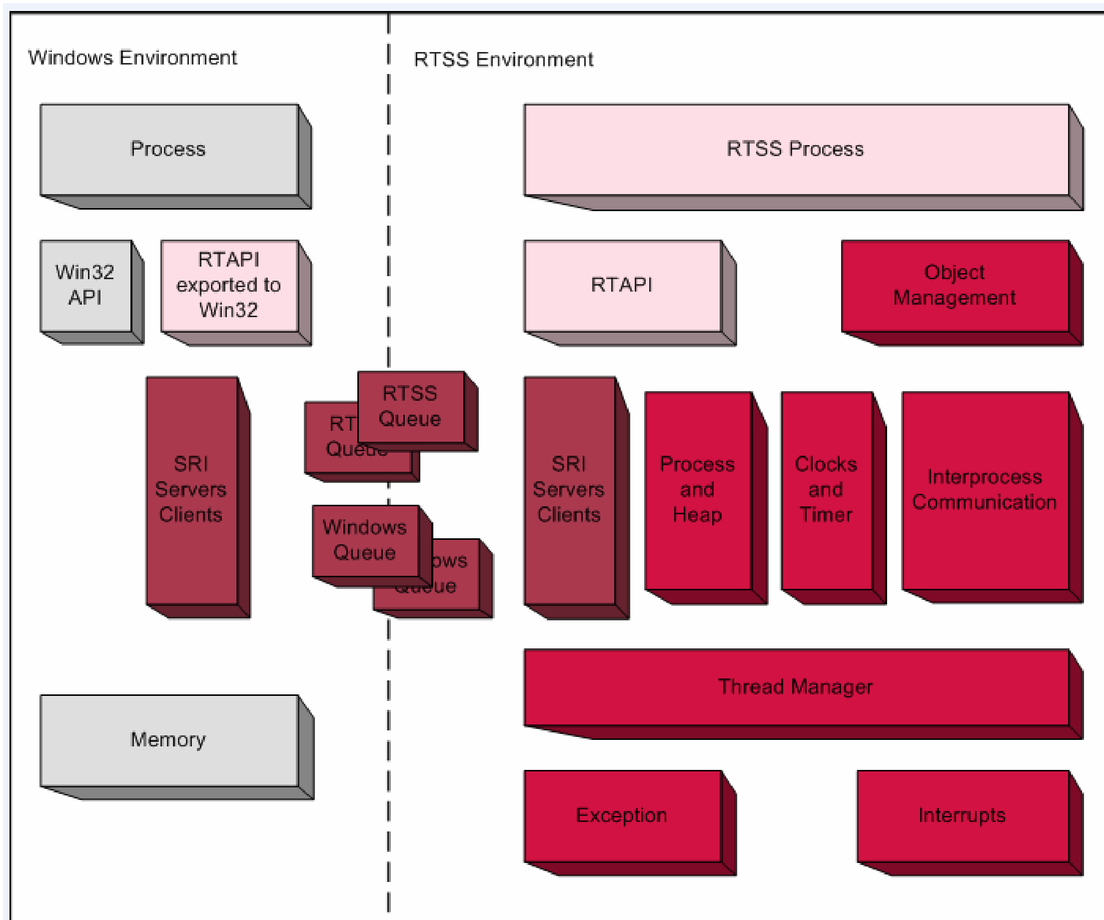


Figure 2. RTSS Detailed Architecture

RTX Memory Management

Out-of-Bounds pointer protection

Although it is recommended to debug the application to the point that there are no bad pointers, there is still a need to have some basic protection. RTX accomplishes this by implementing read-only guard pages around thread stacks to guard against a bad pointer memory write. This mechanism will generate an RTSS “Memory Access Fault” exception that can be handled with structured exception handling (SEH).

Memory Allocation

RTX uses the Windows memory allocator for all of its memory allocation. This is one area where RTX takes advantage of a Windows service to reduce redundancy of functionality. All memory that is allocated to the RTSS subsystem is from the Windows non-paged pool. Because RTX must request memory from Windows, memory allocation is not deterministic. In cases were memory allocation needs to be deterministic RTX supports a local memory pool.

Local Memory Pool

Since RTSS uses the Windows memory allocator, the allocation of the subsystems local pool is non-deterministic. However, once the pool has been created, all memory allocations that use the local memory pool are managed by RTSS and are deterministic.

It is recommended that the memory usage for a given system be characterized in order to optimize the subsystem local pool size before the real-time processing of the applications are started. This will ensure that future memory allocations do not cause more round-trip requests to Windows to grow the pool size.

Large Memory Requirements

RTX operates within the confines of the Windows kernel space and as such there is a limit on the amount of system memory available. This limit is set by Windows and RTX cannot affect it. In the case of many embedded systems, this is not an issue; however, there are systems that require much larger data areas. With RTX, you can map any memory on the system including that memory above the maximum of Windows.

This can also include memory that can only be accessed via Physical Address Extensions (PAE). PAE extends the address space from 32 to 36 bits, allowing a 32-bit system to access memory above 4GB up to 64 GB. The developer must implement their own techniques for how this memory is accessed based on their needs.

RTX Interrupt Handling

The RTX HAL Extension works with the Windows HAL to manage interrupts in order to ensure that RTX interrupt priority is higher than Windows. Because the RTX interrupt is at a higher priority than Windows interrupts, the RTX interrupts will always be available regardless of what Windows is doing at any time. RTX interrupts include the RTX HAL clock tick RTSS interrupt devices (line-based or message-based), and Service Request Interrupts (SRI).

When an interrupt occurs, the RTSS ISR calls the customer's Interrupt Handling Routine and checks if there is a RTSS thread, e.g., Interrupt Service Thread (IST), ready. If there is no RTSS thread ready, the RTSS ISR will restore the Windows context and Windows will continue execution from where it was interrupted.

Message Signaled Interrupts

PCI version 2.2 introduced a new interrupt architecture based on message signaling rather than the standard line-based signaling. This new architecture does not rely on hardware lines and thereby does not have the limitations often associated with running out of resources or inflexibility in configuration of system peripherals. Message signaled interrupts (MSI) use the system's memory to write interrupt messages directly to the interrupt controller.

Although there are devices that support MSI, the Windows XP family of operating systems does not. Windows Vista does support MSI, however, many of the embedded systems running Windows are using a variant of XP. With RTX's support of MSI, embedded designers can take advantage of MSI, even on Windows XP-based OSs. This is a huge advantage when dealing with situations where Windows has ownership of IRQ lines that are needed by a real-time device. Selecting MSI capable devices resolves the conflict without any additional requirements other than the standard conversion process involved in moving a device from Windows to RTX.

RTX and Interrupt Latency

Software Causes of Latency

A switch from Windows 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 Windows interrupt latency. Let us examine the sources of Windows ISR latencies without RTX present.

The most significant cause of latency is IRQ masking by the Windows kernel and drivers, routinely done for periods of up to several milliseconds via Windows *KeRaise/LowerIrql* calls. The Windows kernel, HAL, and certain special drivers also perform processor-level masking of all interrupts via x86 STI/CLI instructions, for periods of up to 10 μsec.

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

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 Translation Lookaside Buffers (TLBs). Video drivers are particularly aggressive users of caches, causing contention-related flushes when an RTX interrupt 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. Also, Intel's® SpeedStep® Technology can cause long latencies when switching between power states.

Some systems, again typically notebooks, use the Pentium processor's System Management Mode (SMM) to perform specialized keystroke or other processing in BIOS firmware. While in SMM, the processor does not field interrupts, which adds to ISR latencies.

Bus mastering events can cause long-latency CPU stalls. Such cases include those initiated by 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 tools to diagnose platform-related latencies, and to 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.

RTX Interrupt Latency Reduction Techniques

RTSS entirely eliminates latencies from IRQL masking by Windows and Windows drivers. The RTX HAL Extension performs interrupt isolation, reprogramming the PIC or APIC when switching between Windows and RTSS. The result is that RTX interrupts can always interrupt Windows, while RTX masks all Windows interrupts when RTSS is running.

Processor-level interrupt masking on PIC systems, on the other hand, cannot 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 to use IRQ locks instead. The RTX HAL extension scans the HAL 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 10 microseconds on a typical Pentium 4, 2.4 GHz system.

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 Windows. The IPC set includes mutexes, events, semaphores, and shared memory objects.

The RTSS object manager uses the Windows non-paged memory pool for its storage requirements. Using Windows-provided mechanisms decreases RTX's resource consumption; however, object allocation is non-deterministic, unless using memory from a pre-allocated local memory pool.

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, or the time quantum (default is infinite) specified for the thread has expired and another thread at the same priority is ready.

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-code routines.

While an RTSS thread is running, all Windows-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 *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.

Service Request Interrupt (SRI)

An important architectural feature of RTX is its lockless interrupt-driven interface between Windows and RTSS. This clean architectural separation has enabled ports of RTSS to various environments (e.g., multiprocessor RTX) ensuring a fast and robust implementation. The Windows 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

Windows-to-RTSS request is an IPC operation like WaitForSingleObject or a Release operation on a RTSS object. A typical RTSS-to-Windows 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.

Win32-RTSS Communication

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.

RTSS Proxy Model

IPC, along with all other Windows-RTSS communication, uses the SRI channel. Given that the SRI channel prevents Windows 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 Windows 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 are worth the tradeoff.

Fast Timer Support

On all platforms the RTX Hal Extension provides clock resolution of 1µsec or better, and timer periods of 100µsec on PIC and 1µsec on APIC systems. If no RTSS applications are executing, then there is no timing difference between real-time HAL and a regular HAL systems.

Dynamically Linked Libraries

No tour of Win32 would be complete without a mention of DLLs. RTSS supports two types of real-time dynamic linked libraries; RTSS DLL and RTDLLS.

RTSS DLLs

An RTSS DLL is the analog of an implicitly loaded Win32 DLL. It is an RTSS process that exports functions for use by other RTSS processes. RTSS DLLs are only loaded into memory one time. Once loaded, they share a common address space with other RTSS processes. RTSS DLLs accurately mirror the automatic resolution of references to exported functions for implicitly loaded Win32 DLLs. The entry point for RTSS DLLs is Main, and they have an associated .LIB file containing information about the exported functions that can be linked with RTSS applications.

There are three primary differences between RTSS DLLs and implicitly loaded Win32 DLLs:

- RTSS DLLs must be explicitly loaded and unloaded by the developer.
- There are no per-process global variables within an RTSS DLL. Since RTSS DLLs are full RTSS processes, all global variables in an RTSS DLL belong solely to the RTSS DLL process.
- Main is used as an entry point, not DllMain. Also, RTSS DLL Main only gets called when the RTSS DLL is initially loaded. Any per process or per thread initialization must be accomplished

through an additional function explicitly exported from the RTSS DLL, and explicitly invoked within the calling RTSS process.

RTDLLs

An RTDLL is the analog of an explicitly loaded Win32 DLL. RTDLLs are RTSS objects that can be dynamically loaded and unloaded using the LoadLibrary and the FreeLibrary calls. They are automatically unloaded from memory when the last RTSS process referencing them terminates. The entry point for an RTDLL is DllMain. Because RTDLLs do not require application to link to an explicit export library, they provide a convenient and flexible way to accommodate changes after a product deployment.

RTDLLs differ from Win32 explicitly loaded DLLs in two significant ways:

- RTDLLs do not have per-process global data; all the static and global data in an RTDLL is shared between all RTSS processes attached to that RTDLL.
- DllMain is called when the last attached process frees the DLL, not once as each attached process frees the library as is the case for Win32.

Structured Exception Handling in RTSS

Structured Exception Handling (SEH) is a rather important feature of Win32 and Windows 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 C Runtime *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 an exception dispatcher similar to user mode code.
- Win32 SEH uses a special interrupt when it sets a new user context after unwinding the stack; RTSS restores context similar to user mode code.
- Windows may edit (move) an exception trap frame with interrupts disabled; RTSS does this similar to user mode code.

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.

RT-TCP/IP

RT-TCP/IP provides deterministic processing of the TCP/IP protocol stack within the RTSS environment and adds networking capability to RTSS applications.

In order to maximize portability between Windows and RTX, RT-TCP/IP provides applications with an interface (API) that conforms to the latest version of the Windows Sockets 2.0 (WinSock).

RT-TCP/IP provides an RTX TCP/IP Protocol Stack that accesses the physical transport layer (network card) via an Ethernet driver running under RTSS.

Performance

The following tables and figures present key performance metrics. All systems were tested with RTX 2009 on XP SP2.

Dell Optiplex SX280 P4 2.8 GHz 500 MB RAM

Operations	Windows Latency (ns)	RTX Latency (ns)
SetEvent (no thread switch): min / max	625/297430	265/1927
SetEvent□ WFSO: min / max	860/332035	397/765
ReleaseMutex□ WFSO: min / max	892/296700	525/620
ReleaseSemaphore□ WFSO: min / max	885/296965	475/612
Yield: min / max	770/335432	230/352
Thread priority change: min / max	775/133320	452/1942
Interrupt service thread dispatch: min / max	600/348000	1000/21000

Dell Dimension 9200 Core2 Duo 2.66 GHz 2GB Ram Shared

Operations	Windows Latency (ns)	RTX Latency (ns)
SetEvent (no thread switch): min / max	387/280812	135/11503
SetEvent□ WFSO: min / max	533/276127	210/605
ReleaseMutex□ WFSO: min / max	560/292409	251/11022
ReleaseSemaphore□ WFSO: min / max	552/276323	236/11048
Yield: min / max	436/283872	127/11131
Thread priority change: min / max	511/282530	225/11635
Interrupt service thread dispatch: min / max	365000/1272000	0/3000

Dell Dimension 9200 Core2 Duo 2.66 GHz 2GB Ram Dedicated

Operations	Windows Latency (ns)	RTX Latency (ns)
SetEvent (no thread switch): min / max	383/273048	135/1244
SetEvent□ WFSO: min / max	533/291868	210/447
ReleaseMutex□ WFSO: min / max	556/286842	259/635
ReleaseSemaphore□ WFSO: min /max	556/274191	240/1093
Yield: min / max	432/67018	127/578
Thread priority change: min / max	507/275943	233/601
Interrupt service thread dispatch: min / max	2000/250000	0/1000

Custom built Core2 Quad 2.4 GHz 2 GB RAM Dedicated between P1 and P2

Operations	Windows Latency (ns)	RTX Latency (ns)
SetEvent (no thread switch): min / max	405/274860	
SetEvent□ WFSO: min / max	558/148687	1590/3986
ReleaseMutex□ WFSO: min / max	596/285112	1841/8538
ReleaseSemaphore□ WFSO: min / max	588/136878	1612/3881
Yield: min / max	457/274470	
Thread priority change: min / max	536/276330	
Interrupt service thread dispatch: min / max	1000/86000	0/3000

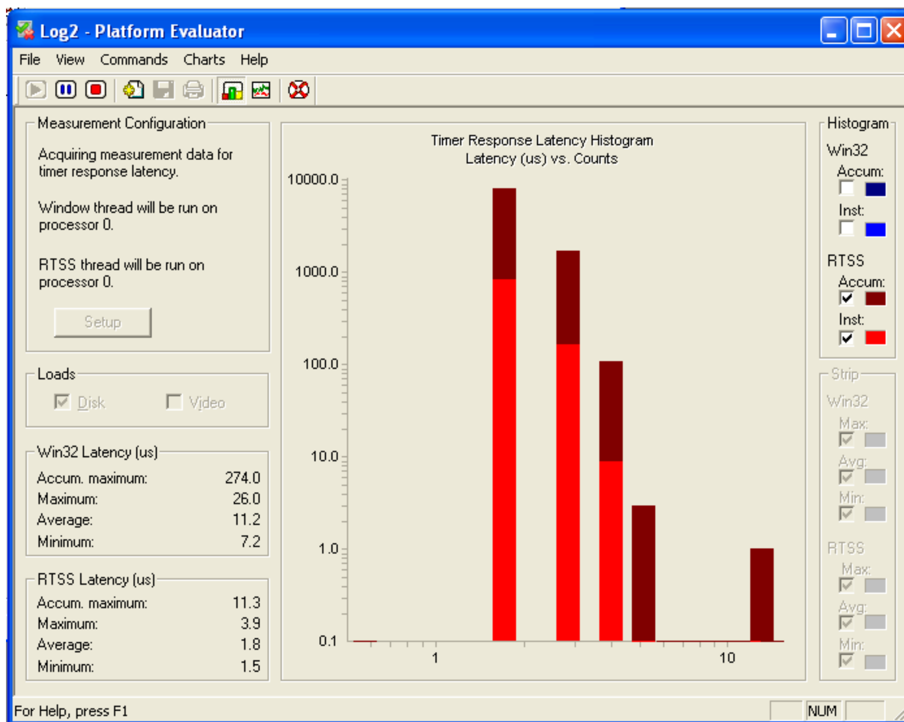


Figure 3. RTSS timer-interrupt latency histogram with a typical Windows workload on a UP system: disk searches, video updates, network activity, etc. The X-axis is in microseconds, and the Y-axis is the number of samples on a logarithmic scale. The lighter bars represent activity during the last second only, while the top of the bar is the accumulation for the total run.

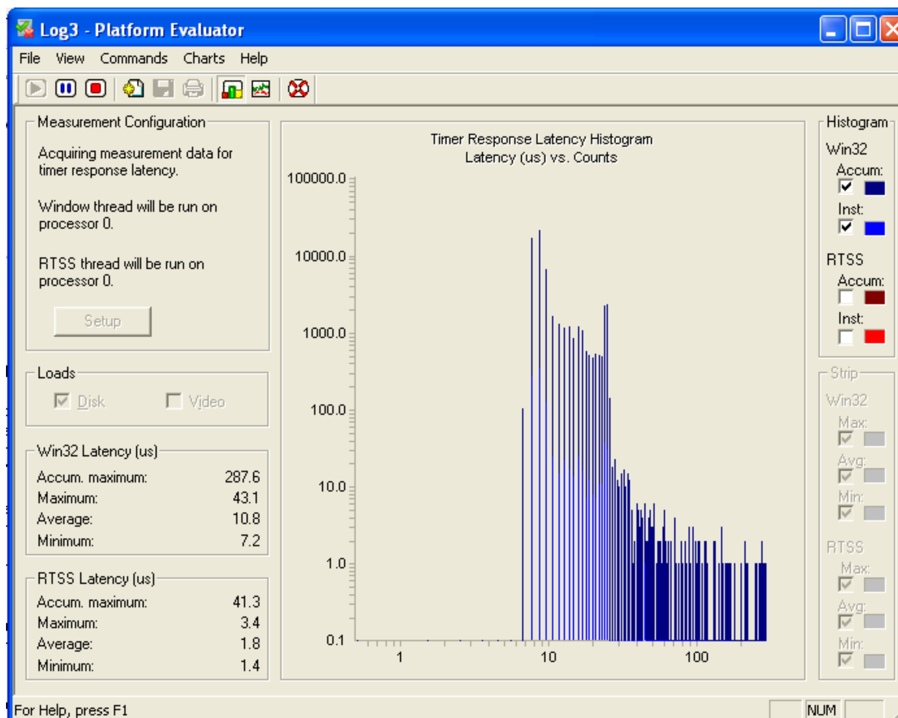


Figure 4. Win32 timer-interrupt latency histogram with the same workload as Figure 3.

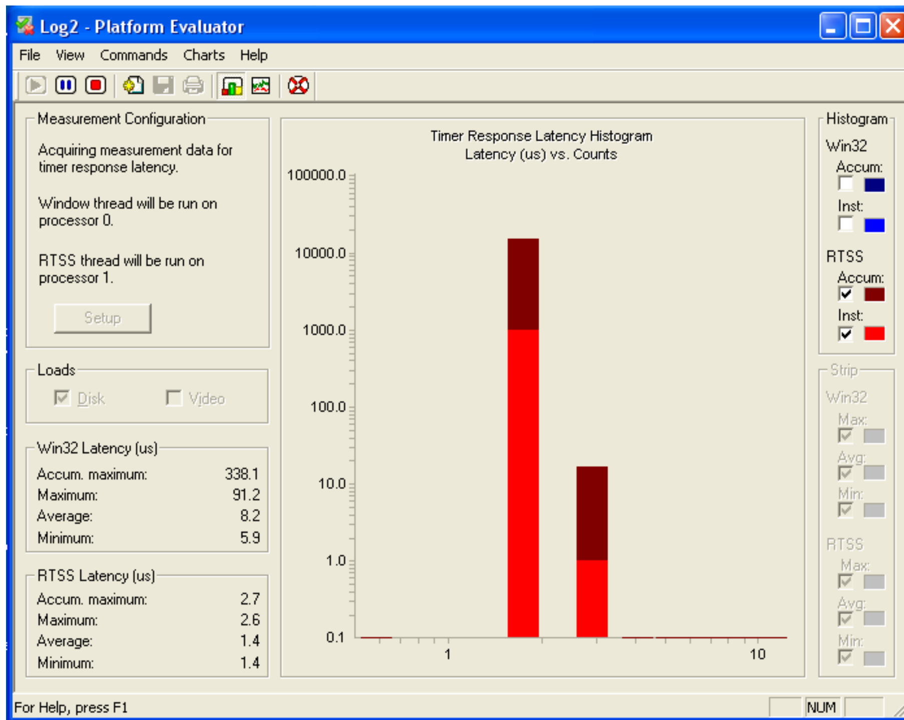


Figure 5. RTSS timer-interrupt latency histogram with a typical Windows workload on an RTX MP dedicated configuration system: disk searches, video updates, network activity, etc. The X-axis is in microseconds, and the Y-axis is the number of samples on a logarithmic scale. The lighter bars represent activity during the last second only, while the top of the bar is the accumulation for the total run.

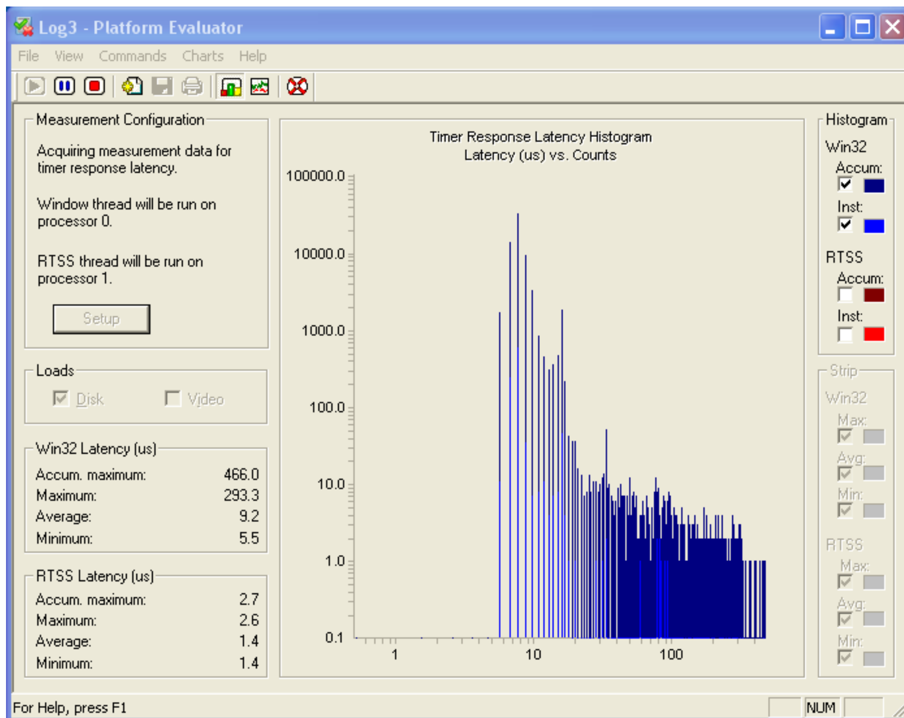


Figure 6. Win32 timer-interrupt latency histogram with the same workload as Figure 5.

Performance Tools

RTX performance tools enable developers to qualify and tune performance on their platforms.

- Kernel System Response Time Measurement (ksrtm) tool 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 Windows component or device driver is causing the greatest latency event: after such an event, the ksrtm Interrupt Service Routine (ISR) obtains the address of the interrupted sequence from the stack then resolves it to a loaded Windows kernel module.
- System Response Time Measurement (srtm) tool is a simple RTAPI timer latency measurement tool running either in RTSS or Win32, which produces a histogram, to realistically reflect timer latencies.
- Latency Test Measurement (ltp) tool determines long-latency events due to bus mastering events.
- PerformanceView is a tool that displays system CPU usage by RTX applications, Windows applications and the system idle process. This tool will also provide the maximum duration Windows was starved each data collection period when RTX is run in shared mode.
- Platform Evaluator is an easy-to-use tool for determining the right hardware platform for your real-time application. It helps developers, system integrators, and vendors to evaluate their real-time control platforms and select the best hardware, software, and operating system configuration for their particular needs. Platform Evaluator collects system configuration information and measures thread switch latencies, timer interrupt response latencies, and throughput time of selected API calls in a variety of user-specified test environments.

Development Tools

RTX provides developers with the ability to easily create and debug their RTSS applications.

- Wizards are provided to allow for easy creation of RTSS applications, device drivers, and NIC configuration in Microsoft Visual Studio 2008, 2005, .NET 2003.
- RTX provides host-target debugging support for Microsoft Visual Studio 2005 and .NET 2003, along with local debugging support for Visual Studio 2005, .NET 2003, .NET 2002, and 6.0.
- Microsoft WinDbg kernel level debugger is extended to allow developers to see RTSS processes and thread along with IPC Objects.
- RTSS Object Viewer is a graphical tool that provides state information about all processes, threads, and IPC objects within the RTSS environment.
- TimeView is a real-time event tracing tool that allows developers to efficiently capture and display the execution sequence of threads within RTX. TimeView is designed to minimize the intrusion on the real-time task being monitored. An easy-to-use graphical user interface provides both a Data Collection Setup Wizard and a data viewer.

Conclusion

IntervalZero's RTX has shown that it is possible to augment Windows 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 that 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.intervalzero.com. The product is available for free evaluation and all evaluations are accompanied by a Quick Start video.

Copyright © 2009 IntervalZero, Inc. All rights reserved. All trademarks, trade names, service marks and logos referenced herein belong to their respective companies
Ref#DOC-RTX-004
June 2009