Extending Windows XP into real time

By Paul Fischer

The use of Windows in the embedded marketplace continues to grow, acknowledged directly by Microsoft's aggressive promotion of two Windows platforms for the embedded market: Windows CE and Windows XP Embedded (XPe). CE addresses the small and mobile marketplace, whereas XPe is a specially packaged version of Windows XP with some features intended for complex embedded applications. This article looks at the impact of dual and multicore microprocessors with a combined Windows XP and Real-Time Operating System environment.

Standard Windows XP is a rich platform on which to build complex applications. The familiar user interface allows one to build embedded instruments that can be quickly mastered by the end-user. Modern PC-based computers can display complex images and interface with a wide variety of networking protocols.

Windows XP Embedded (XPe) adds several key features that are crucial to OEMs for the use of Windows in an embedded application:

- Simpler licensing terms than standard Windows
- Custom configuration of Windows (by removal of unnecessary components)
- *Hibernate once, resume many* and avoidance of *first time install* messages
- *Gold disk* manufacturing process for more efficient factory floor production techniques

Avoiding *first time install* messages, the ability to build a fully configured system at the factory floor with a single unique image, and removal of unused Windows features are all key reasons for choosing XPe. Even though a system is based on Windows, it is not required that the entire Windows environment be exposed to the end-user. Windows XPe accommodates removal of those parts of the system that are either unnecessary or, by their inclusion, might compromise the usability or safety of the system. In other words, XPe provides a means to constrain the embedded Windows device in order to operate strictly for the function(s) for which it was intended.

XPe's unique *hibernate once, resume many* feature insures a consistent initial system state, even following an unplanned system shutdown (such as pulling the plug). Startup is rapid and can commence with the primary device application loaded and ready for use, taking less than 10 seconds (following completion of the BIOS POST) in a typical system.

Real-time software requirements

A *real-time system* is one in which the time to respond to an event is as important as the logical correctness of the response. Typical hard real-time systems require a level of determinism and performance that can be measured in the tens of microseconds.

Windows XP Embedded is a tool for developing a managed OS feature set. This is critical for applications in which the components of the operating system must be known and controlled. Despite all of its advantages, the XPe tools do not address determinism. The number of processes allowed to run can be limited by XPe, but the system's overall lack of real-time determinism cannot be fixed.

Windows XP and XPe are capable of providing fast overall response to events; however, neither is appropriate on its own for applications that require true hard real-time determinism. A Windows application thread, regardless of its priority, can be preempted by one of many software and hardware sources – including interrupts and high-priority kernel and driver threads.

Real-time software must also be robust and reliable. While a programming error in a business application might lower the productivity of the user, an error in a real-time control program can result in costly downtime, damage to expensive equipment, or even the loss of human life. Tools and protection mechanisms must be made available to the real-time developer to aid in minimizing the occurrence of typical programming errors such as stray pointers, memory leaks, and uninitialized variables. In the event of faulty code and/or a software crash, protection mechanisms minimize the impact of a software crash on the critical processes being controlled.

Why Windows XP is not real-time

Windows is built around a fully preemptive, multitasking kernel that is capable of satisfying, on average, an application's timing and response requirements. This means that Windows will miss some scheduling deadlines, and will be late for many other deadlines. Whether or not this condition is acceptable depends upon the requirements of the application.

Windows will miss or be late for deadlines due to some fundamental kernel policies. These policies were put in place for a very good reason: to optimize the average system performance by being fair to all applications on the machine. These policies spell trouble for the real-time systems programmer; real-time applications are not fair, and a real-time application must be selfish with regard to its use of the processor when compute cycles are needed.

Determinism (the ability to meet deadlines predictably) is what separates real-time applications from general-purpose applications. A deterministic application can only be built if the timing of events managed by the operating system, upon which it relies, is reliable and predictable and the developer is allowed extensive control over the relative priorities of all operations and events. Windows restricts the ability to control and predict threads and events in a number of ways:

- Because the Windows priority spectrum places all interrupts at a higher level than normal thread execution, user-level threads will always be preempted by any interrupt source, regardless of the importance of that interrupt. This means that even moving a mouse will generate an interrupt that preempts a high priority non-interrupt operation. Only kernel-level threads can raise or lower the Interrupt Request Level (IRQL) to mask or unmask interrupts. In configurable Real-Time Operating Systems (RTOS), thread priorities can be interleaved with interrupt priorities, giving the developer total control over the relationship between interrupts and threads.

- Windows utilizes a Delayed Procedure Call (DPC) mechanism to increase the responsiveness of the system to interrupts. A correctly designed Interrupt Service Routine (ISR) minimizes interrupt latency by performing only critical processing in the ISR and then queuing a DPC for further processing as a thread. DPCs are placed into a single FIFO, with no provisions for the priority of the operation. This means that a low priority DPC will execute first, regardless of the priority of other DPCs queued behind it. You can cheat and place a DPC at the head of the queue, but this does not solve the problem because you may inadvertently be deferring the execution of another higher priority DPC that is already in the FIFO. Additionally, since DPCs are lower on the priority spectrum than all other types of interrupts, DPCs will not execute before interrupts – even low priority interrupts.

- Multiple requests for a synchronization object in Windows (such as a semaphore or a mutex) are also queued in a FIFO, again without regard for the priority of the requesting thread. Thus a high priority thread may have to wait for a low priority thread to complete its operation before proceeding. This affects not only determinism, but can also lead to priority inversion.

- Solving the classic problem of priority inversion requires the ability to inherit the priority of another thread, which is not available in Windows (Windows uses a random boosting mechanism that does not always solve the problem in adequate time). The inversion problem can be described as follows: assume at least three threads, A, B, and C, with A being the highest priority and C the lowest. Priority inversion occurs if C has previously locked a resource and A is waiting on that resource, but C is unable to complete its job because it is has been preempted by B. Thread A has been effectively held off by lower priority thread B. Temporarily boosting C's priority to A's (C inherits A's priority level) remedies the problem.

Virtual hardware via software

To balance the flexibility of Windows with the deterministic requirements of embedded applications, designers usually add a dedicated real-time component (in effect, a second computer). This *dual-computer dual-OS* solution is built upon two distinct hardware computing elements: one computer system to run Windows and a second system to host a dedicated RTOS.

Unfortunately, a second control computer adds substantial cost of goods, manufacturing complexity, and system-to-system coordination headaches. A *single-computer dual-OS* system, where one compute element hosts both the Windows system and the RTOS, significantly reduces the cost of goods and complexity, and simplifies the coordination of Windows with real-time processes. It can also reduce design costs, measured in the time and effort spent on engineering tools and staff, which alone may be sufficient reason to seek this simpler solution.

This might seem like an unrealistic proposition, but it is possible for one hardware platform to support two operating systems using virtual machine technology. By running the RTOS in one of the virtual machines, a *dual-computer dual-OS* architecture is converted into a *single-computer dual-OS* solution. However, in order for such a solution to be considered a viable option it must, at the very least, satisfy the following criteria:

- The RTOS must be safe, secure, reliable, and extensible
- The hardware platform must support dependable, hard real-time functionality
- The application must be easy to design, build, and integrate using standard tools

Applications built for a *single-computer dual-OS* system must be partitioned into deterministic and nondeterministic parts. The nondeterministic application executes on the Windows virtual machine, and the deterministic application executes on the real-time virtual machine. The two appli-cation parts must have managed access to a variety of shared objects (such as mail-boxes, semaphores, mutexes, shared-memory) to work together properly.

Real-time processes and threads running on the RTOS virtual machine need access to high-speed interval timers for accurate, low-drift time measurements and for gen-erating exact periodic intervals to insure precise control of real-time systems. x86 Advanced Programmable Interrupt Controller (APIC) uniprocessor systems and multi-processor systems, the vast majority of embedded and desktop Windows platforms built today, are excellent candidates. The accuracy and drift of the timer elements in x86 APIC systems is very good.

The ability to use a single standard development tool for both environments is a significant advantage of the virtual machine. The real-time environment needs direct access to I/O and memory, a fixed priority scheduling system with priority-inversion protection, and simplified interrupt-handling services to insure efficient implementation of real-time threads. This allows developers to create and deploy sophisticated real-time applications without having to write complex and cumbersome device drivers for Windows XP for access to real-time hardware.

The virtual machine *single-computer dual-OS* approach eliminates redundant computer and communication hardware and improved communication and coordination between the real-time and Windows applications.

Dual-core enhancements

The process described previously, using two virtual machines to share a single CPU platform that supports Windows and real time, works for a large number of real-time Windows applications. Typically, applications with cycle times of one millisecond or slower are served quite well by this arrangement and have been deployed on the current crop of desktop and industrial motherboard platforms (uniprocessor and hyper-threaded Pentium 4 class processors running at 1-3 GHz). There are, however, some applications that demand faster cycle times. For these applications there is another solution that is just now possible with the introduction of dual-core processors into the mainstream.

This alternate solution retains the cost and efficiency benefits of the virtual machine *single-computer dual-OS* approach, but no longer requires sharing the CPU and key processor resources. When virtual machines share a CPU, as is the case with the single-core processor designs common today, they must maintain a full machine context (or a partial context in the case of a hyper-threaded core) in order to switch between the two operating systems. Saving and restoring these contexts results in some compromises regarding event response latencies and maximum cycle times. These compromises typically contribute on the order of 10 to 30 microseconds to interrupt jitter. For cycle times of one millisecond or slower 10-30 microseconds of interrupt latency represents a jitter variation of only a few percent.

Higher speed cycle times mean higher bandwidth controllers, a desirable trait because it leads to improved performance and throughput. However, 10-30 microseconds of jitter is a significant number when cycle times of 50-200 microseconds are needed. If jitter is a significant percentage of the cycle time it adversely affects the quality of the control algorithm. The stability and quality of the controller are a function of the accuracy of the cycle time. Variations in cycle times degrade the stability margin of a closed-loop control system, especially naturally unstable systems like position-feedback motion control loops.

Dual-core processors can support two operating systems by dedicating one CPU to the RTOS. The CPU instruction cycles of the dedicated core are available 100 percent of the time to the RTOS. The CPU cycles of all remaining cores become the exclusive property of the Windows virtual machine. Contention for key CPU resources such as pipelines, cache, and the FPU are avoided. Using the built-in interprocessor communication mechanisms accomplishes coordination between the two processors, eliminating context switch times entirely. In this scenario, interrupt latencies are reduced by an order of magnitude, from 10-30 microseconds down to 1-3 microseconds. Loop cycle times in the 50-200 microsecond range can operate with high precision and accuracy. The advent of inexpensive dual-core hardware means an order of magnitude in quality and bandwidth control algorithms can be implemented on a real-time Windows platform.

In addition to vastly decreased jitter, a dedicated RTOS processor in a dual-core system has the advantage of 100 percent dedicated CPU cycles. When two operating systems share a single CPU they must be willing to share the raw compute performance in order to assure that both operating systems will be able to execute and perform the tasks at hand. In other words, if one OS consumes all the CPU cycles the other OS will effectively be *frozen out*. When a CPU can be dedicated to each OS this is no longer a concern. On a dual-core system the real-time tasks can maximize the number of compute cycles they consume (because they can consume 100 percent of their CPU's cycles and Windows will still run). This means that even more complex control algorithms can be developed, resulting in further increases in the quality and performance of the control systems that can be implemented on real-time Windows systems.

Safety and reliability by design

Note that the virtual machine approach to adding real time to Windows is quite different from installing a real-time kernel in the form of a Windows device driver or subsystem. The device driver and subsystem models force real-time applications to operate as part of the Windows kernel. Kernel mode code has privileged

"The ability to use a single standard development tool for both environments is a significant advantage of the virtual machine."

access to the entire memory space, including the Windows kernel and other device drivers; it lacks address isolation and memory protection. A real-time thread running on such a system can easily overwrite other processes, both real-time and Windows processes. Because such programming errors are difficult to detect in kernel mode and result in spurious but critical failures, achieving reliable operation through this method often requires extensive testing and debugging, with many errors not detected until the system has been deployed in the field. Creating a complex, multi-threaded, real-time application to run inside the Windows kernel is contrary to the notion of building reliable, safe, and dependable real-time applications.

By definition, when Windows crashes (for example, blue screen) something catastrophic has occurred and Windows itself cannot recover. The integrity of all of Windows is in question, including interrupt handling, the kernel, and device drivers. Thus, continued operation of a real-time driver or subsystem that is encapsulated within the Windows kernel, following a Windows crash, is unreliable at best, and will likely result in failure of the real-time subsystem and all of its processes.

Using a virtual machine to add real time to Windows means the RTOS will maintain reliable operation of real-time processes in the event of a Windows crash. The virtual machine approach to real-time Windows allows real-time applications to run in user-mode, not kernel-mode. The result is improved reliability and robustness, as well as simplified programming and debugging. Each real-time process built on a virtual RTOS runs in a separate 32-bit protected memory segment. These segments are distinct from those used by Windows and provide address isolation and protection not just between the real-time processes, but also between real-time processes and non-real-time Windows code.

Real implementation – INtime for Windows

TenAsys Corporation has implemented such a system: the INtime RTOS running in a virtual machine alongside Windows. The INtime RTOS virtual machine makes it possible to extend Windows applications into the real-time domain by providing a separate hard real-time virtual machine on which the real-time components of an application reside. A complete real-time Windows application consists of both non-real-time Windows processes and threads, and real-time processes and threads. Real-time processes typically handle time-critical data acquisition and control, while non-real-time processes handle the human interface, network communications, and data storage.

The complete *single-computer dual-OS* system consists of the following key components, as illustrated by Figure 1.

Standard Windows XP

Because the system is divided into two virtual machines, there is no need to utilize a special version of Windows; a standard Windows distribution can be utilized (such as Windows XP, XPe, Windows 2000). Non-real-time processes and threads execute normally on the Windows kernel. Off-the-shelf applications can be used without change; they are unaware of the real-time kernel.

Real-time kernel and API

The real-time kernel provides deterministic scheduling and execution for real-time processes and threads. Real-time interrupts and threads preempt the execution of all Windows threads and disable non-real-time interrupts. Real-time applications utilize the real-time API to access the capabilities of the real-time kernel.

NTX, API, and DLL

The NTX interface provides a mecha-nism for communication between the Windows virtual machine and the INtime virtual machine. NTX provides applications with managed access to a variety of shared objects, such as mailboxes, semaphores, mutexes, and shared-memory.

OS Encapsulation Mechanism (OSEM)

The INtime OSEM manages the virtual machines to insure simultaneous opera-tion and integrity of the Windows kernel and the real-time kernel. This virtual machine mechanism provides memory protection and address isolation be-tween all Windows processes and real-time processes.

Shared development environment

Because the system includes Windows, it is not necessary to have a separate development workstation; the target system is also the development system. Standard Windows development tools, such as the Microsoft Visual Studio tools, can be used for both the Windows part and the real-time part of an ap-plication. A single compiler, linker, editor, and debugger serve both virtual machines.

Conclusion

When applying Windows XP to time-critical applications, a real-time operating system is necessary to satisfy the requirements for accurate and repeatable data acquisition and control. An RTOS that shares the CPU with Windows, using virtual machine technology, allows embedded Windows applications to take full advantage of the Windows' standard user interface, network capabilities, development tools, and off-the-shelf software and still deliver the performance required by critical, hard real-time tasks.

Because the virtual machine approach to integrating an RTOS with Windows does not require special hardware, it does not result in increased hardware complexity. Software development is also simplified because the real-time developer can use the same Microsoft Visual Studio development environment used to build standard Windows XP applications.

The virtual machine architecture insures that Windows-based real-time systems always perform data acquisition and machine control at the highest priority; with overall supervisory control and display of data on the user interface running at the lowest priority. Real-time events always run at the highest priority level, regardless of the version of Windows or the hardware platform used.

TenAsys Corporation provides INtime Real Time Extensions for Windows, a fully featured RTOS derived from iRMX. Applications written for the INtime RTOS execute with guaranteed determinism as fully protected user-mode processes in concurrence with the

Microsoft Windows operating system on standard PC hardware. Because real-time application code executes in user-mode, the INtime environment is immune to application faults that crash kernel-mode driver solutions.

Using the INtime real-time operating system extension for Windows, designers can reduce the cost of implementing computer-based control systems by utilizing x86-based PC hardware and Microsoft Windows software. INtime software combines the benefits of the Windows operating system – standard APIs, networking, user interface, and development environment – with a proven, highly reliable, real-time kernel designed for critical applications. ECD

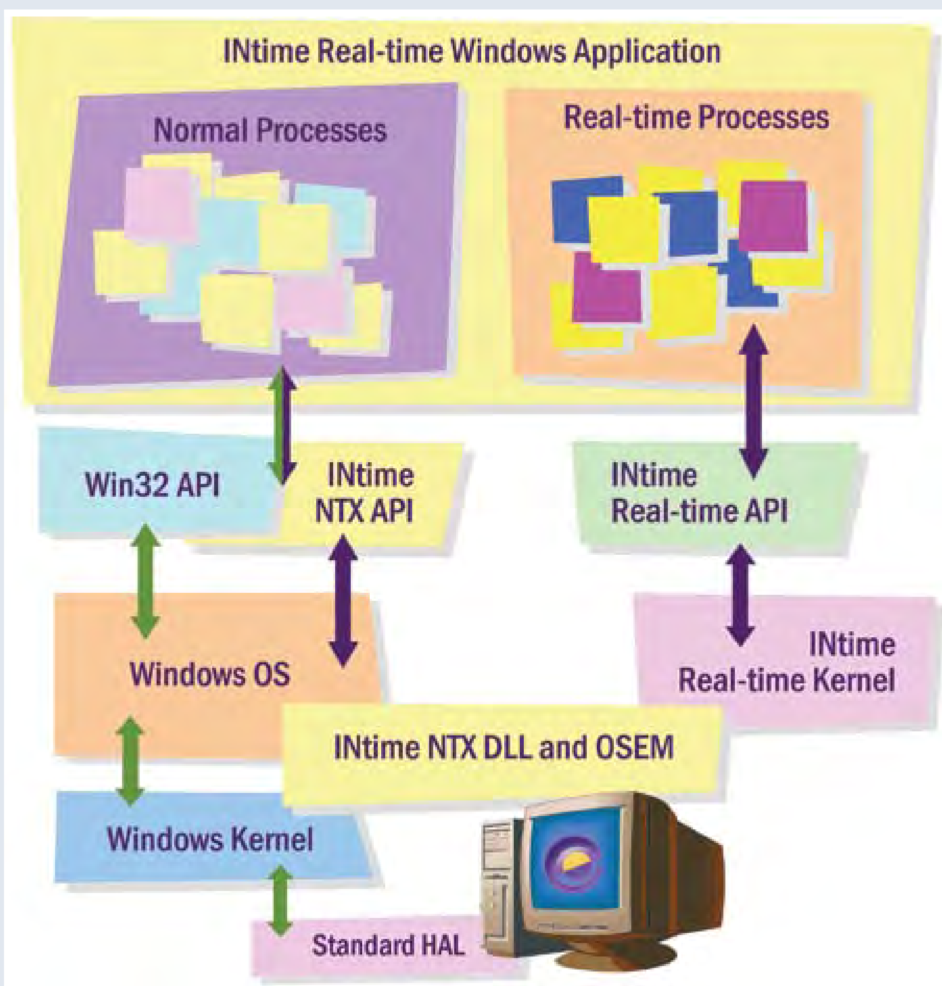
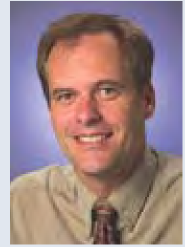


Figure 1



Paul Fischer *is a senior technical marketing engineer at TenAsys Corporation. Paul's experience with INtime goes back to 1997, when the product was first introduced. He has more than 20 years experience with real time and embedded systems in a variety of engineering and marketing roles. Fischer has an MSE from UC Berkeley and a BSME from the University of Minnesota.*

To learn more, contact Paul at:

TenAsys Corporation
1600 NW Compton Drive, Suite 104
Beaverton, OR 97006
Tel: 503-748-4720 • Fax: 503-748-4730
E-mail: info@tenasys.com
Website: www.tenasys.com