

A Comparison of Approaches

A variety of products are available today that support the integration of hard real-time programming environments with Microsoft's Windows NT operating system (OS). In most instances the goal of these products is to provide developers with a convenient way to leverage their knowledge of Microsoft Win32 application programming interfaces (API) and tools so they may create deterministic applications which will run in conjunction with standard, off-the-shelf, Win32 applications. The nature of the real-time execution environment and the philosophy applied to the real-time API are often referenced when comparing these products. Both of these characteristics affect the ability of a Win32 developer to create bug-free, reliable, real-time applications.

INTRODUCTION

Real-time software requirements

The deterministic nature of a real-time system dictates a different set of requirements when compared to a business application or inventory management system. A simple definition for a real-time system is one in which the time required to respond to an event is just as important as the logical correctness of that response. Hard real-time systems have the strictest requirements and need the highest level of determinism and performance. Their worst case event response times can be measured in the tens of microseconds.

Robust and reliable operation

Real-time software also must be robust and reliable. A programming error in a word processing program simply lowers the productivity of the user of that application; an error in a real-time control program can result in costly downtimes, damage to expensive production equipment, or even the loss of human life. Tools, APIs, software protection mechanisms, and the program execution environment must be designed to help the real-time developer minimize the likelihood of programming errors such as stray pointers, memory leaks, and uninitialized variables, as well as aiding the developer in the process of locating errors in a program's logic. In the event of a failure due to faulty code (a.k.a. a software crash) the real-time execution environment must be able to minimize the impact of the software crash on critical processes.

A clear and concise real-time API

No less important a requirement, the real-time API must be clear and concise in order to avoid unnecessary confusion in the use of that API; confusion that could result in difficult to locate bugs. An API, which is just a list of function names, does not adequately describe the programming environment; the real-time execution environment in which the application runs is also relevant. The ability to support features such as virtual addressing and multiple independent processes in the real-time execution environment has a significant impact on the type of real-time system a developer can implement and the tools they can employ to create their real-time application.

APPROACHES TO REAL-TIME WINDOWS NT

Notwithstanding the limitations of Windows NT as a deterministic operating system, there is tremendous promise for the application of Windows NT to real-time applications. The fundamental approach being used today, by several real-time vendors, to bring the advantages of Windows NT to embedded, real-time systems is to integrate a real-time kernel (RTK) within standard Windows NT.

Clean integration with the Windows NT environment, that is, leveraging the Windows NT development tools and APIs, is critical for the general usability of the integrated real-time Windows NT solution. The definition of what is meant by a real-time API that leverages the Microsoft Win32 API is a hotly debated topic. What is meant by saying that an API is Win32-compliant? Is there a "right way" to implement a real-time API that will be used in conjunction with Windows NT applications and development tools? What impact does the real-time execution environment have on compliance with standard Win32 programming practices?

Integrating a real-time kernel with Windows NT

Any solution that claims to bring hard real-time performance directly to a Windows NT system must provide an alternate kernel to perform the scheduling and execution of real-time tasks. In fact, the three major hard real-time solutions for Windows NT available today have taken this approach. In such a system an RTK runs in conjunction with the standard Windows NT kernel.

Introducing an RTK into the Windows NT environment can decrease system reliability, unless the kernel is at least as reliable as the Windows NT kernel. It is critical, therefore, that the RTK be proven in real-life applications, with extensive testing that can only come through repeated use over time. Other important considerations to this approach are memory and address space protection for real-time applications, as well as the ability to survive catastrophic system failures (commonly referred to as Windows NT "blue screen" crashes).

Approaches to Integrating a Real-Time Kernel

There are at least two approaches in use to couple an

RTK with the Windows NT kernel: 1) place the RTK inside a Windows NT interrupt service routine (ISR) or device driver or 2) place the RTK completely outside of Windows NT's address space in a separate and distinct 32-bit x86 hardware task. The differing characteristics of these approaches have a profound impact on the nature and reliability of the real-time execution environment. The semantics of the real-time API and the execution environment presented by the RTK are affected by the approach taken to integrate the RTK.

User-mode versus kernel-mode operation

At first glance, putting an RTK inside a Windows NT ISR or device driver would appear to be the most straightforward and easiest implementation. However, this approach requires that the end-user develop real-time applications in Windows NT's kernel-mode (a.k.a. ring-0 or privileged-mode). In contrast, standard Win32 applications always run in Windows NT's user-mode (a.k.a. ring-3 or protected-mode). The differing characteristics of these two operating modes are enforced by microprocessor hardware and are found on virtually all 32-bit microprocessors in use today.

Ring-0 lacks sufficient memory protection and is difficult to debug

Software that executes in Windows NT's kernel-mode can access the entire memory space of the system, including the Windows NT kernel and other device drivers; there is no hardware-enforced address isolation or memory protection available to software that runs in Windows NT's kernel-mode. Thus, a real-time application executing in a kernel-mode environment can overwrite the address space of another Windows NT application, the Windows NT kernel, or another real-time application. Because these types of programming errors are very difficult to detect, and often result in spurious but critical failures, achieving reliable operation requires extensive testing and debugging. Many errors may not be detected until after the real-time system has been deployed in the field. Writing a complex, multithreaded real-time application in kernel-mode is contrary to the programming model encouraged by Microsoft for Win32 applications.

The x86 hardware task environment maximizes real-time integrity

Equally serious is the challenge associated with maintaining reliable operation of the RTK in the event of a Windows NT "blue screen" crash. By definition, when Windows NT "blue screens" it has detected a catastrophic software failure from which it cannot recover. Following a blue screen crash the integrity of the entire Windows NT environment is in question, including ISRs, device drivers, and Windows NT kernel services. Continued operation of an RTK that is an integral part of the Windows NT kernel space will be unreliable at best, and may ultimately lead to a crash of the real-time processes running on such an RTK.

Patent pending OSEM Technology Improves Fault Tolerance

RadiSys' patent pending Operating System Encapsulation Mechanism (OSEM) is responsible for the simultaneous operation of Windows NT and the INtime RTK on the same CPU. It insures that real-time responsiveness is maintained regardless of any Windows NT activities. This encapsulation approach utilizes the hardware included in every 32-bit x86 microprocessor (Intel, AMD, and others) to implement the maximum level of address space isolation and software environment protection possible between non-real-time processes and real-time processes.

INtime hardware tasks are transparent to Windows NT

In a standard Windows NT configuration the bulk of the OS runs within the confines of a single x86 hardware task. Additional hardware tasks are defined only to handle catastrophic failures, such as stack faults and double faults, where a safe, known environment is required from which to handle the failure. INtime transparently creates an additional x86 hardware task for the RTK and manages the process of switching between the standard Windows NT hardware task and the real-time hardware task. This approach guarantees the integrity of both the Windows NT kernel and the RTK, and insures continuous operation of real-time processes even in the event of a total Windows NT failure. The OSEM adds a new level of fault tolerance to Windows NT. By placing critical processes under the control of the INtime RTK, one is able to guarantee

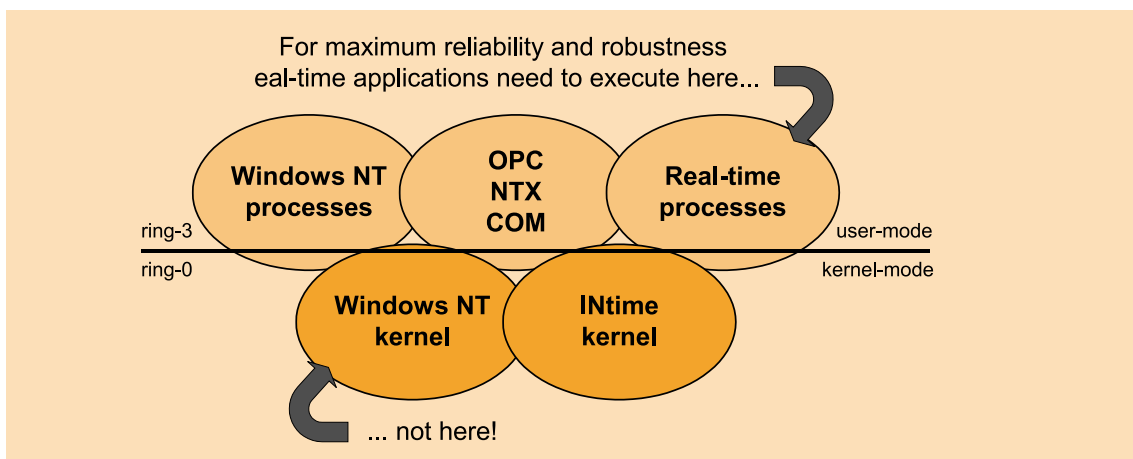


Figure 1: user-mode execution environment for INtime real-time applications

continuous operation despite any failure of Windows NT applications or of the Windows NT operating system.

The OSEM minimizes interference with Windows NT

Because the OSEM encapsulates the entire Windows NT priority spectrum as the lowest priority INtime thread your real-time threads and interrupts are guaranteed to always have priority over Windows NT threads. This means that real-time code always operates deterministically, regardless of any parallel Windows NT activities. INtime's OSEM provides a clean, well defined interface that minimizes interaction with Windows NT to a few key areas, resulting in improved product reliability.

The INtime OSEM does not rely on knowledge of either the Windows NT kernel-mode environment or the Windows NT device driver environment. Standard Windows NT device drivers can be added and removed to a Windows NT system with no impact on the ability of the OSEM to operate. Thus, changes in the Windows NT's device driver run-time environment, which could affect the operation of a device driver-based implementation RTK for Windows NT, have no affect on the OSEM and its ability to support the execution of real-time applications. This INtime encapsulation mechanism also simplifies compatibility with future releases of the Windows NT operating system.

The OSEM eliminates the need for kernel-mode programming

Because the OSEM creates a separate, protected, user-mode environment for real-time processes, real-time application developers are relieved of the burden of writing and debugging code in Windows NT's kernel-mode. The result is improved reliability and robustness, as well as simplified programming and debugging. The INtime RTK automatically creates a separate 32-bit protected memory environment for each real-time process, just as Windows NT does for each of its 32-bit applications. INtime 32-bit protected memory environments are separate and distinct from those created by Windows NT and includes additional address isolation and protection mechanisms that exceed the protection provided for Windows NT process environments. This address isolation exists not only between multiple real-time processes but also between real-time processes and Windows NT processes. No special tools or programming rules are required of the developer to take advantage of these features, this memory protection is supported automatically using standard Windows NT development tools (e.g., Microsoft Visual C/C++).

REAL-TIME API PHILOSOPHIES

Embedded real-time processes have some unique requirements that differ from those of a standard Windows NT process. Some of these differences are expressed by the need for an API that is specific to the development of real-time applications. Other differences are found in the nature of the run-time environment. As described in the previous section, each INtime process is loaded into and runs in a distinct 32-bit protected-mode memory environment which

behaves very much like the protected-mode memory environment created by the Windows NT kernel for standard Windows NT applications.

Applying the process model to real-time applications

Address protection without restricting access to I/O

In the INtime real-time execution environment, the address space delegated to each real-time process is unique (each occupies a separate virtual address space) and any attempts to use addresses not assigned to a process will be trapped by the real-time kernel. Unlike the Windows NT process environment, the INtime RTK does not restrict access to I/O devices (using the IN and OUT instructions), attachment of threads to interrupt service routines, or manipulation of the processor's interrupt flag (using the STI and CLI instructions). The INtime real-time API includes system calls that can be used to access memory-mapped I/O devices via standard C/C++ pointers with dynamically assigned virtual addresses. These relaxations in the restrictions of the real-time process environment are necessary to reduce the overhead associated with accessing hardware from a real-time environment. They also result in a significant simplification of the programming effort needed when real-time code must access an I/O device.

A multiple process model that supports multiple threads is a perfect emulation of the Windows NT programming environment

The concept of multiple processes, each capable of containing multiple threads of execution, and the address isolation that only the protected-mode process model provides, is accomplished by executing applications in user-mode. Without such a runtime environment it is very difficult to reproduce the type of execution and debug environment to which Win32 programmers have become accustomed and expect to be present. For example, INtime's protected user-mode environment allows developers to debug applications knowing that bad pointers will be reliably trapped by the kernel, using the protected-mode hardware built into every 32-bit x86 Intel architecture microprocessor, without the fear of inadvertently writing over the data or code in another process space. It also means you can develop complex real-time applications without concern for where a process is located in physical memory relative to other processes, since each protected-mode process is guaranteed to have a unique and consistent addresses space, even if multiple copies of the same application are loaded!

Integration of a real-time API with the Windows NT API

Integration with the Windows NT environment is the key to success for any real-time Windows NT solution. This means leveraging the standard Windows NT development tools and APIs wherever possible, in order to minimize the developer's learning curve and maximize their efficiency.

Multiple philosophies for real-time Windows NT

Two basic philosophies exist regarding the creation of a real-time Win32 API for the development of real-time

Ad RadiSys

Windows NT applications. One philosophy suggests that the real-time API should identically match the existing Win32 API syntax. We believe that this approach is prone to confusion and that a Win32 real-time API must be defined to use Win32 naming, parameter passing, and error return conventions with a syntax that matches the semantics of the functionality required of the real-time execution environment.

What is a "Win32 API"?

As a point of reference, consider what the term "Win32 API" really means. Microsoft does not maintain one static definition of the Win32 API for use by Windows NT, Windows 95, and Windows CE applications. It is a constantly changing and evolving entity. With each new set of application requirements Microsoft will extend the Win32 API to address new needs in functionality; they do not "back fill" the existing Win32 API with new semantics. The DirectX, OpenGL, IrSock, WinSock, WinINET, and WNet APIs are just a few examples of recent additions to the Win32 API that have either extended or replaced an existing Win32 API subset or have added a completely new dimension to the Win32 API which previously did not exist.

INtime defines a "perfect-fit" Win32 API for real-time Windows NT applications

This is exactly the philosophy taken for the INtime Win32 real-time API. Since Windows NT does not, by design, support deterministic real-time applications it is necessary to augment the Win32 API in order to provide the functionality required by hard real-time applications. By creating an API that follows standard Win32 naming conventions, type definition rules, and error return procedures with a syntax that matches the semantics of the real-time problem domain, the developer easily recognizes the programming environment and is not confused by the functionality behind the API.

For every real-time extension it is necessary to guarantee that real-time applications limit themselves to the real-time API in order to preserve their deterministic behavior. A distinct API that follows Win32 standards and conventions is the best way to guarantee that subtle bugs are not introduced by any confusion regarding the definition of a function in the real-time API.

Real-time API syntax and semantics

A real-time API that is syntactically identical to the

Microsoft Win32 API but semantically different (i.e., differs in its behavior) only serves to confuse the developer. The goal behind having syntactically identical APIs should be to give the programmer the message that identically named calls work exactly as the original function calls on which they are based. Unfortunately, this goal cannot be achieved without differences in the behavior of the syntactically identical functions. Why? Because Microsoft's Win32 API and execution environment was never designed to provide hard real-time services to real time applications. In order to provide deterministic, real-time functionality one must deviate from the standard Win32 API. Doing otherwise is misleading because it results in creating syntactically equivalent real-time functions to Microsoft's Win32 API which differ in functionality from their same-named non-real-time counterparts.

A Specific Example: CreateThread() and SetThreadPriority()

The CreateThread() and SetThreadPriority() functions are excellent examples of the difficulty associated with applying existing Win32 semantics to a hard real-time environment. After "creating" a thread with CreateThread() it is still necessary to specify the thread's priority using SetThreadPriority(). Microsoft's version of this function allows you to specify one of seven relative priority levels; that is, a thread's priority is specified to be relative to the base class of its parent process. Specifying a relative priority is appropriate for a non-real-time thread running in a non-deterministic environment but is not sufficient for a deterministic thread that must maintain an absolute, hard real-time priority. To maintain the identical syntax paradigm, in such a case, an identically named function must be used which interprets the nPriority parameter differently (i.e., as an absolute priority with more priority levels) or a slightly different function must be used which has an interface like Microsoft's SetThreadPriority() function but incorporates these same subtle differences in its functionality. The values which may be assigned to the problem parameter in either of these two functions, nPriority, and the meaning of that parameter, results in significant semantic differences between the original Win32 function and the nearly identical real-time function.

```
HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpThreadAttributes,    // security attributes
    DWORD dwStackSize,                          // initial thread stack size in bytes
    LPTHREAD_START_ROUTINE lpStartAddress,      // pointer to the new thread
    LPVOID lpParameter,                        // arguments for the new thread
    DWORD dwCreationFlags,                     // creation flags
    LPDWORD lpThreadId,                        // pointer to returned thread
                                              // identifier
) ;
BOOL SetThreadPriority (
    HANDLE hThread,                             // thread handle
    int nPriority                                // thread priority level
) ;
```

Figure 2: Win32 CreateThread() and SetThreadPriority() system calls

The Importance of Priorities

Setting the priority of a standard Windows NT thread has vastly different meaning than setting the priority of a hard real-time thread. In the Windows NT environment you must choose among seven base priorities which the Windows NT's scheduler may then further modify by boosting or demoting. The Windows NT scheduler modifies the base priority of a process in order to implement "fairness" when scheduling ready-to-run threads. Windows NT thread priorities are always specified as being relative to their process' base priority. Since the Windows NT scheduler may boost or demote the base priority level of a process you cannot exercise absolute control over a thread's priority level within the context of the entire system.

Fairness and real-time do not mix

In order to accommodate deterministic, hard real-time behavior, real-time environments must offer more than just seven priority levels, and they must allow you to specify an absolute priority on a per-thread basis. The scheduler (promoted or demoted) cannot arbitrarily manipulate thread priorities for fairness reasons. They may, however, be modified temporarily to implement priority inversion schemes, which are necessary to avoid deadlocks. The concept of setting a hard real-time priority is very different from that of the Windows NT model of priorities. Attempting to represent hard real-time parameters and behavior by using an API in a real-time environment that is identical to that defined by the standard Win32 environment is misleading and prone to subtle programming errors.

The Security Attributes Parameter

All real-time extensions to Windows NT are based on the implementation of a separate real-time kernel. In order to maintain the real-time kernel's reliability and determinism it is necessary to minimize its interaction with the Windows NT kernel. A real-time Win32 API that supports Windows NT kernel structures like the `lpThreadAttributes` parameter in the `CreateThread()` call will only result in compromising the determinism of the real-time application environment. Thus, any real-time API that includes such structures must ignore them. Including unused parameters ultimately means confusion for the developer because their presence implies support. To insure this confusion does not result in latent bugs it is best to simply remove the parameter from the API.

Missing Functionality

The Win32 API supplied by Microsoft cannot adequately support real-time applications. Many key functional elements are missing. Real-time applications typically have a need to precisely measure time intervals, access specific physical memory addresses, temporarily enable and disable interrupts, and creating in-process interrupt handlers which can easily interact with other threads in the same process. A real-time API must add those functions that address these missing elements in order to satisfy the needs of real-time applications. Clearly, the a real-time Win32 API should be defined so that it encompasses the needs of real-time applications, rather than attempting to extract

functionality from an API that was never intended to provide that service.

CONCLUSION

The approach taken by RadiSys to define the INtime real-time Win32 API results in a clean and obvious separation between the hard real-time execution environment and the non-deterministic Windows NT environment. INtime's real time API follows established Win32 naming, parameter typing, and error return standards and conventions but does not attempt to use an identical naming and parameter interface for functions which are similar but not identical to those provided by Microsoft's Win32 API. Likewise, the protected-mode real-time execution environment provided by the INtime extension very closely matches that provided by Microsoft's Win32 subsystem for standard Win32 applications. Our perfect-fit real-time API minimizes developer confusion and maximizes productivity by promoting the use of standard Windows NT tools and programming conventions without compromising the functionality required of a hard real-time application.

When looking to Windows NT as a platform for embedded real-time systems, it is critical to assess what the application will demand in terms of reliability and determinism, both now and in the future. Making an informed decision requires understanding the fundamental mechanisms of Windows NT and how they affect the utility of the operating system in a real-time environment and the basic approaches to solving this problem.

By utilizing proven real-time technology and providing seamless integration with Windows NT, RadiSys has made reliable real-time Windows NT a reality. Corporations can utilize industry standard Windows NT across the entire organization, from desktop and business applications to high-end embedded applications such as telecommunication and data communication equipment, all the way down to the factory floor. These embedded real-time applications can take full advantage of the standard Windows NT user interface, its powerful network capabilities, rich development tools, and off-the-shelf software, and still deliver the rock solid, reliable performance required of real-time systems. ■

Paul Fischer joined the RadiSys software group in August of 1997 to help design and market their INtime real-time extension for Microsoft Windows NT. Previously, Fischer spent seven years at Force Computers in various engineering and marketing roles related to the application of software to embedded systems and, most recently, as the chair and editor of the BusNet VMEbus Backplane Protocol committee sponsored by the VITA Standards Organization (VSO). He has more than fifteen years experience with real-time embedded systems serving in a variety of engineering and marketing roles for Parker-Compumotor, Seagate Technology, and several Silicon Valley start-ups. Fischer has an MSE from UC Berkeley and a BSME from the University of Minnesota.