

Real-time Extensions to Windows NT Are they right for your next real-time project?

With the recent introduction of real-time extensions to Windows NT, many real-time developers are starting to consider NT for their next project. It's easy to see why. Rather than connect real-time and desktop applications over a network, it appears that developers can now integrate both into a single system, while using a single API.

But is NT with real-time extensions really a solution for mission-critical real-time applications? Let's look at the capabilities we have come to expect from a real-time operating system (RTOS) - such as determinism, reliability, low overhead, and source-code portability - and see how "real-time NT" compares.

REAL-TIME DETERMINISM

By definition, real-time applications are required to respond to external events within predictable time limits. This is especially true of "hard" real-time systems, where missed deadlines can have dire, or even disastrous, consequences.

A real-time system's ability to respond to external events within a specified time in known as determinism. To indicate how well an RTOS can support determinism, most vendors quote at least the following performance metrics (see Figure 1):

- **interrupt latency:** the time from the start of the physical interrupt to the execution of the first instruction of the user-written interrupt service routine (ISR)
- **scheduling latency:** the time from the execution of

the last instruction of the user-written interrupt handler to the first instruction of the process made "ready" by that interrupt

- **context-switch time:** the time from the execution of the last instruction of one user-level process to the first instruction of the next user-level process

While these metrics don't provide a full indication of an RTOS's determinism, they can help you assess whether an RTOS can achieve the determinism and performance required for your real-time application. Just as important, they can help you compare the performance of a real-time NT extension to that of a native RTOS.

OS vendors tend to quote these metrics across a range of processors. For example, let's look at the figures for QNX, a real-time operating system used in a wide variety of real-time applications. Times are in microseconds (see table 1).

How do NT real-time extensions compare? The numbers vary, but the published figures for some extensions indicate performance numbers 10 to 15 times slower than the numbers in table 1.

Why are these extensions so much slower? One rea-

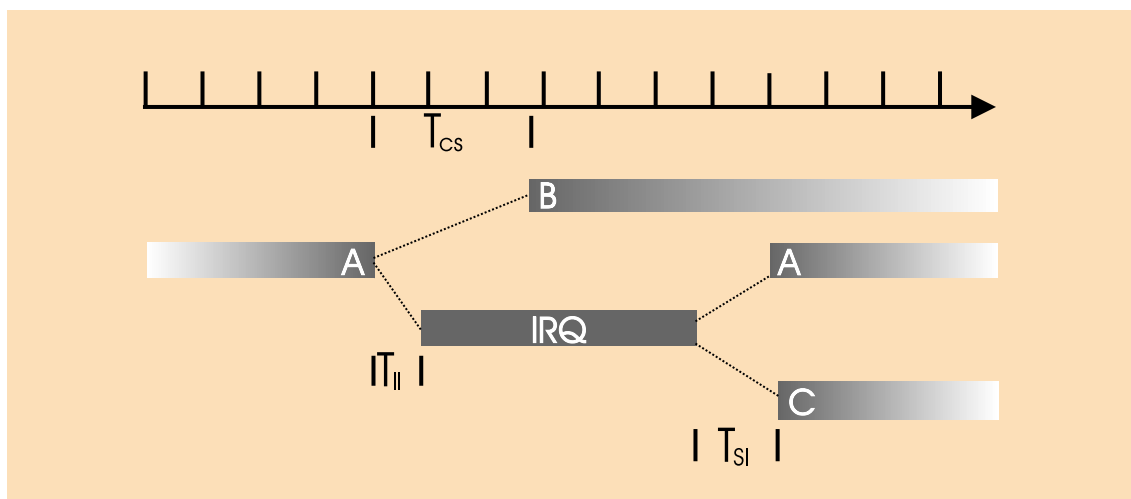


Figure 1. To provide an indication of an operating system's real-time determinism, most OS vendors refer to the following metrics: interrupt latency (T_{II}), scheduling latency (T_{SI}), and context switch time (T_{CS}).

PROCESSOR	INTERRUPT LATENCY	SCHEDULING LATENCY	CONTEXT SWITCH
Pentium 200	1.4	2.9	1.2
Pentium 100	1.8	4.7	2.6
486 DX/33	7.5	12.6	8.2

Table 1.

son is that they repeatedly poll a hardware interrupt in order to give control to the real-time subsystem. While determinism could be improved by increasing the polling rate, this increase uses up CPU cycles that your NT applications may require to achieve acceptable performance. The problem becomes worse in a networked application since network cards not only require access to as many CPU cycles as possible but also impose their own high interrupt rate.

HIGH AVAILABILITY AND ROBUSTNESS

Determinism is important, but there are additional criteria for measuring a real-time system, such as high availability. Can the system's OS continue to run - or at least recover rapidly - if a software fault occurs? For that matter, can the OS continue to provide services even if a critical hardware component, such as a hard drive, fails?

Achieving high availability is a complex problem that requires a variety of features in the OS. For example, let's consider how the OS deals with software faults.

No matter how hard we try to write error-free code, a practical reality is that our real-time applications will contain undetected programming errors, such as stray pointers and out-of-bound array indices. Any of these can cause a software fault and, potentially, cause the system to crash. To detect such errors, you need an OS that supports the Memory Management Unit (MMU) found on most of today's 32-bit processors. If a memory-access violation occurs, the MMU will notify the OS, which in turn can abort the errant process at the offending instruction.

Some real-time extension products for NT provide memory protection for real-time processes; some do not. But even if an extension supports memory protection, you still have to ask whether it will let you implement a software watchdog.

What is a software watchdog? It's a process that is informed by the OS whenever a memory violation occurs. This process then makes an intelligent decision on how to recover from the fault.

HARDWARE VS. SOFTWARE WATCHDOGS

To understand the importance of a software watchdog, let's look at what many existing systems use to recover from software faults: a hardware watchdog timer attached to the processor reset line. Typically, a component of the system software checks for system integrity, and then strobes the timer hardware to indicate that the system is "sane." If the hardware timer isn't

strobed regularly, it expires and forces a processor reset. The good news is that the system recovers from the software or hardware lockup. The bad news is that the system must also completely restart, which defeats our goal of high system availability.

Compare this behavior to a software watchdog, which can intelligently choose from several, less drastic, recovery methods. Instead of always forcing a full reset, the software watchdog could:

- simply restart that process without shutting down the rest of the system, or
- abort any related processes, initialize the hardware to a "safe" state, and restart the related processes in a coordinated manner, or
- if the failure is critical, perform a coordinated shutdown of the entire system and sound an audible alarm to notify the maintenance staff

The software watchdog lets you retain programmed control of the system, even though several processes within the control software may have failed. A hardware watchdog timer can still help you recover from hardware "latch-ups," but for software failures you now have much better control. Furthermore, by employing the "partial restart" approach, your system can survive intermittent software failures without experiencing any downtime.

AN OBVIOUS CHOICE

While performing a partial restart, your system can also collect information about the nature of the software failure. For example, if the system contains or has access to mass storage (flash memory, hard drive, a network link to another computer with a hard drive), the software watchdog can generate a chronologically archived sequence of process dump files. These dump files can then give you the information you need to engineer a "fix" before you experience similar failures.

A software watchdog not only decreases costly (or even dangerous) downtime, but also helps you avoid software faults in the future. For these reasons, you should make sure a real-time NT extension has the features required to let you implement a software watchdog.

REDUCING KERNEL FAULTS

Of course, programming errors don't occur only in application code. To support new hardware or system services, you may need to develop device drivers and other system-level services.

In traditional OS architectures, these components run

Ad SBS GreenSpring

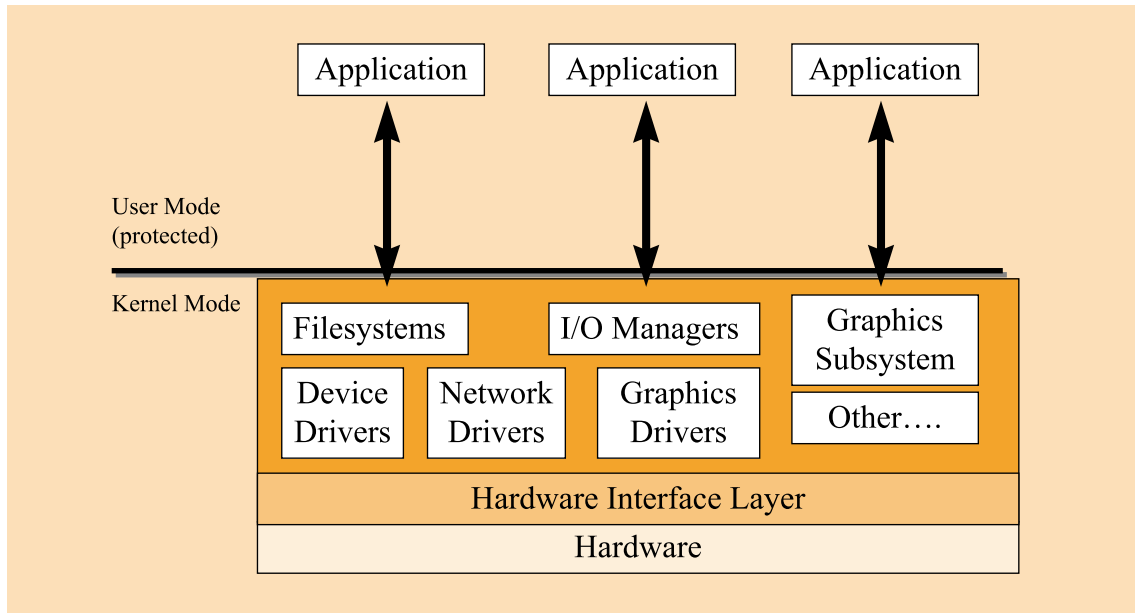


Figure 2. Traditional OS architecture

as part of the kernel - in kernel mode (see Figure 2). Code running in kernel mode runs without MMU protection. As a result, errant pointers or array subscripts in device drivers can cause kernel faults, which only a hardware reboot can remedy. The more code built into the kernel, the greater the likelihood of kernel faults. In Windows NT, these faults result in the "blue screen" crash.

In a microkernel OS like QNX, only the kernel (32k of code) and interrupt service routines (ISRs) run in kernel mode, drastically reducing the possibility of kernel faults (see Figure 3).

The "Blue Screen" Crash

All vendors of real-time NT extensions have recognized the need to deal with blue screen crashes. As a result, some of these products can trap a kernel fault so that

the real-time subsystem can choose to continue running or to close down gracefully. Still, the ability to continue running is a questionable benefit if you can't interact with the NT components of the system--such as the operator interface!

And there is a greater problem: some real-time extensions to NT can potentially contribute to kernel faults. These extensions are implemented directly into the kernel as an interrupt service routine (ISR) or into the Hardware Abstraction Layer (HAL). As a result, the whole real-time subsystem runs in kernel mode. So what happens if you have a stray pointer in your real-time application? You get kernel faults - the blue screen crash.

Also, most real-time applications require custom device drivers. Since all NT device drivers reside in the

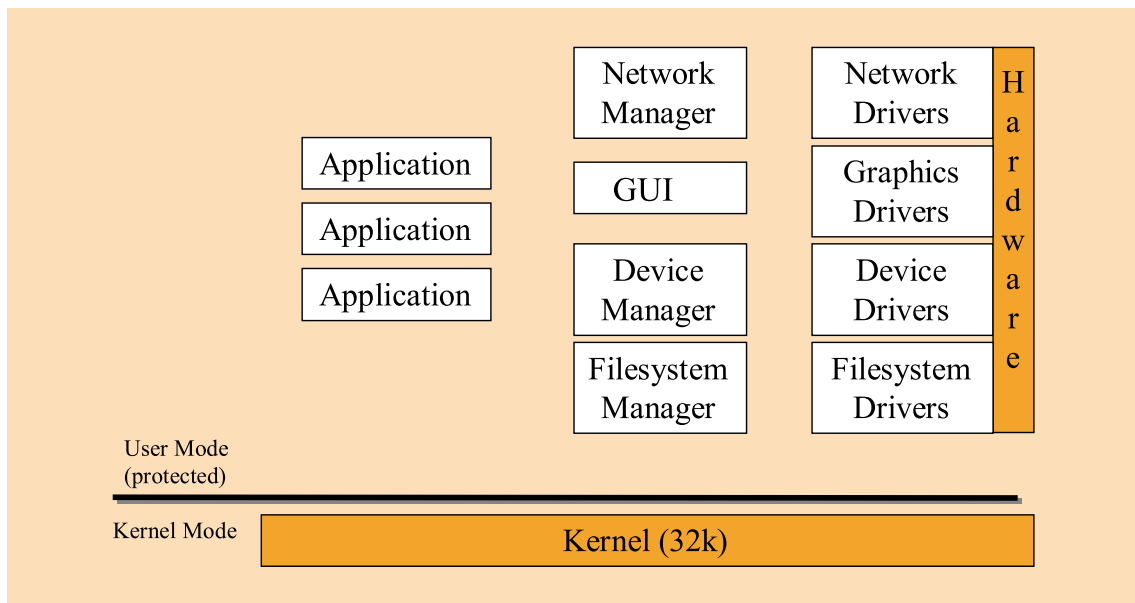


Figure 3. QNX Microkernel Architecture

kernel space, this only contributes to the fragility of the system.

TECHNICAL SUPPORT

The subject of software crashes raises another question: Whom do you call for support when you experience problems? Microsoft, or the vendor of your real-time extension? Before you invest in an extension, you need to determine who will assume the responsibility of providing you with technical support if your system experiences problems.

ACCESS TO RESOURCES

To provide streamlined access to system resources (e.g. filesystems, devices, communications gateways), traditional RTOSs provide an API that is implemented either by system processes or by the kernel itself. A distributed RTOS, such as QNX, goes a step further and turns a network of computers into a single logical machine. As a result, a process running on any computer can, with appropriate privileges, access all resources on the network, including:

- filesystems (hard drives, CD-ROM drives, etc.)
- communications ports (serial, parallel, modems)
- CPUs
- communications gateways (e.g. TCP/IP)

This distributed approach can significantly enhance system availability: If a device fails on one machine, you can automatically restart a process to use a device, or even a filesystem, on another machine.

When evaluating a real-time NT extension, you need to determine whether it will let you access resources from both NT applications and real-time applications. For example, let's say your real-time subsystem requires high-performance access to the NT filesystem. Does the NT extension provide the functionality to let you do this? If so, how does it provide this access? Does it go through the HAL? If it does, you'll end up using the same mechanisms that make NT unsuitable for real time (i.e. you'll lose control over the priority of Deferred Procedure Calls initiated by an ISR). Also, what happens if Microsoft decides to make changes to the HAL? Will your real-time extension stop functioning?

All the above questions also apply to accessing communications gateways.

SYSTEM OVERHEAD

As I mentioned earlier, some real-time NT extensions implement real-time determinism by means of a high-frequency polling interrupt. This interrupt imposes a processing overhead even when no real-time work is to be done. The result? Fewer CPU cycles for non-real-time applications and increased latency. In comparison, most RTOSs are event-driven, responding to interrupts only as they occur.

As for memory overhead, most extensions simply increase NT's already large memory requirements. Most RTOSs, on the other hand, can fit easily into small, ROM-based embedded systems.

CONFORMANCE TO STANDARD APIS

To protect their code investment, many developers strive to create applications that are portable across OS platforms. Industry standards such as the POSIX API have emerged to help developers achieve this goal - even NT offers a POSIX option. Nevertheless, the widespread success of Microsoft operating systems has created an additional, de facto standard: the Win32 API. Consequently, several RTOSs now support both POSIX and Win32.

Unfortunately, some NT real-time extensions support neither POSIX nor Win32. Instead, they use a proprietary API that defeats any goal you may have of achieving platform portability and vendor independence. Other extensions provide only a subset of the Win32 API, and may thus limit the functionality you can implement in your real-time subsystem.

CONCLUSIONS

Real-time extensions to NT offer a degree of real-time determinism that NT alone cannot provide. But having a degree of determinism is only a piece of the puzzle. A real-time environment must also be extremely reliable. It must be able to recover quickly from software faults, without downtime, and avoid kernel faults. For most applications, the environment should impose low CPU overhead and minimal memory requirements. And it should offer a portable API.

As we've seen, many real-time extensions to NT can't fulfill these requirements. Most RTOSs, on the other hand, offer established technologies that have been fine-tuned to the demands of the real-time marketplace. As a result, a "loosely coupled" approach still makes the most sense for most real-time applications: use NT for the desktop, an RTOS for the real-time control, and integrate the two systems via the various networking options now offered by RTOS vendors. ■

Greg Bergsma graduated from the Canberra College of Advanced Education, Australia in 1982, majoring in computer science, electronics and microprocessors. After spending 10 years in the gaming industry, working on real-time transaction processing and embedded gaming terminals, Greg joined the Australian distributor of QNX Software Systems Ltd. - Computer Network Systems also known as CNS. During his five years at CNS, Greg headed technical and business sections with a hands-on approach, providing training, consulting, and contract services to CNS customers. As a result, Greg was involved in the development of a variety of demanding real-time applications, from plastics injection molding systems to children's television game shows. In December of 1996 the desire for more professional and personal challenges led Greg to QNX headquarters in Kanata, where he is now a senior technology analyst. An avid golfer and cricket player, Greg has adjusted to the northern winter by taking up ice hockey and baseball - with varying degrees of success.