The RTX Real-Time Subsystem for Windows NT

Bill Carpenter, Mark Roman, Nick Vasilatos and Myron Zimmerman
VenturCom, Inc.
Cambridge, MA

The RTX Real-Time Subsystem for Windows NT

Bill Carpenter, Mark Roman, Nick Vasilatos and Myron Zimmerman
VenturCom, Inc.
215 First St.
Cambridge, MA 02142
{carp, marcus, boxer, myron}@vci.com

Abstract

This paper describes a subsystem for the Windows NT 4.0 Operating System which implements a kernel-mode execution environment for Win32 compatible tasks and threads that have hard real-time performance characteristics (deterministic interrupt response and dispatch latencies). This subsystem is a proper OS extension which requires no modifications to the standard OS kernel and limited modifications to the NT Hardware Abstraction Layer (***HAL***). This gives the motivation for the approach, describes the design and evaluates the success of the implementation in the context of other strategies for extending general purpose OS kernels.

1. Introduction

Real-time features have been moving from special purpose operating systems into general purpose operating systems for at least 10 years. VenturCom has produced a succession of real-time UNIX™ versions going back as far as 1984. Other implementations are described in [Furht et al. 91 and IEEE 93].

These were responses to the extremely reasonable desire to leverage the features of the underlying general purpose operating system in real-time applications development. This included both accessing the rich feature set of the general purpose OS and accessing available off the shelf applications, utilities and services in developing the particular real-time application at hand.

This is even more so with Windows NT based applications. The system services are rich, diverse and wildly popular with the programming cognoscenti. The available base of applications is vast and growing rapidly.

This is about how real-time computing can be facilitated for a particular general purpose operating system – Windows NT 4.0 with the addition of a subsystem that supports the execution of very low latency real-time threads with Win32 compatible system services. This approach reflects lessons learned about the costs of massive source code whacking – and the benefits of modularity vis a vis the separation of functionality in systems engineering.

2. Real-time Extensions

Requirements for real-time extensions were developed in a process of consultation with company partners whose applications are concentrated in industrial automation and telephony domains. These however span nearly the full range of functionality traditionally associated with hard real-time systems. Windows NT 4.0 has significant real-time features but gating limitations as well [Sommer 96, VENTU 96, Timmerman & Monfret 96, MICRO 95]. The problems most often cited are priority inversion, the paucity of real-time thread priorities and unbounded system response times.

It was further clear from the consultation process that real-time extensions to the system should conform as closely as possible to the standard interfaces of the host OS. It was therefore stipulated that operations in common between real-time objects and normal objects should be accessible via a common interface.

This lead to the definition of a subset of the Win32 API which includes all basic execution control, memory management, communications, synchronization, I/O and configuration operations to be supported by the added real-time threads and to the definition of added interfaces for real-time operations (stack/heap pre-allocation and locking, interrupt and i/o port attachment, clocks and timers) to be provided for both real-time and normal objects. The result is a unified – Real-time Application Programming Interface (RTAPI) with which application elements intended for both the normal Win32 environment and the extension real-time environment can be designed – and with which design and code can be shared.

The implementation of these extensions spans two components. The first implements the real-time operation extensions for normal Win32 objects. The second implements the Real-Time Subsystem (***RTSS***). Both exploit limited modifications developed for the NT HAL (which primarily provide extensions for clock and timer device programming and interrupt management).

Figure 1 illustrates how the Real-Time Subsystem interfaces to NT. The HAL and the RTSS driver provide the Real-Time Application Interface (RTAPI) to real-time processes, which are linked and loaded as NT drivers. The difference between an NT driver and the real-time process is that the real-time process uses RTAPI calls instead of the NT device driver interface.

2.1 Thread Scheduling

The real-time thread manager offers 128 thread priorities and controls priority inversion. The threads are scheduled by priority and within a priority, in round robin order. There is no sharing of processors based on a fixed time slice. All RTSS threads must give up the processor by waiting, changing thread priority or otherwise completing execution. All RTSS threads run before any NT threads can run.

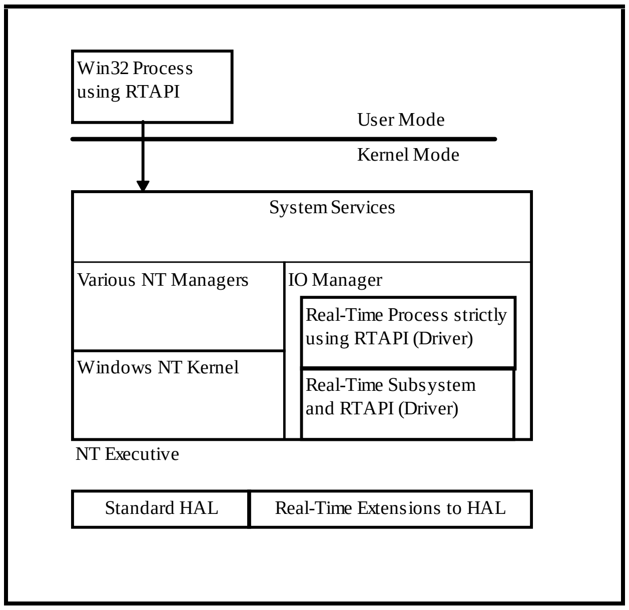
Win32 Process using RTAPI
User Mode
Kernel Mode
System Services
Various NT Managers
IO Manager
Real-Time Process strictly using RTAPI (Driver)
Windows NT Kernel
Real-Time Subsystem and RTAPI (Driver)
NT Executive
Standard HAL
Real-Time Extensions to HAL

Figure 1 Real-Time Subsystem

The thread manager gets control of the processor in response to interrupt processing by the HAL. Two of these interrupts, the clock and the RTSS software interrupt are fixed interrupt sources. The clock interrupt handler handles timer expirations and the RTSS software interrupt causes the RTSS to examine the queue of messages from NT. The last type of interrupt is from a device for which an RTSS process registered an interrupt handler.

The net effect of these interrupts is that some number of RTSS threads will become ready to run. When all of the threads have finished their immediate work, the thread manager becomes idle, the RTSS switches back to the HAL stack and normal NT processing occurs.

2.2 Interrupt Handling

Interrupt objects are used to claim system interrupt resources and handle interrupt events. The interrupt object is an abstract type which captures the resource usage and the service routine associated with an interrupt.

All interrupt service routines in RTSS are realized as real time threads. In designing the system in this way, a small performance tradeoff was made. For a given interrupt, the handler latency increases to include thread switching time. This is a relatively small increase in latency, however. The benefit to this approach is that all real time system activities are captured as real time threads, which provides a simple, coherent approach to setting priority across the real time subsystem, and allows for a more robust and verifiable implementation of the subsystem.

Interrupt masking is achieved through lazy or optimistic interrupt masking. This technique manages the interrupt level in software to reduce the overhead of hardware programming. Actual hardware masking occurs only in the event of an actual hardware interrupt, at which point the mask is set to prevent further interrupts, and the hardware event is noted so that it can be dealt with at the end of the critical code section. This technique is described in [Stodolsky et al. 93].

2.3 Communication with NT

In order for NT and the RTSS to communicate, two message queues are maintained by the RTSS. One queue is for messages passing from NT to the RTSS and the other queue is for messages passing from the RTSS to NT. The queues are implemented as circular buffers of messages. This organization is necessary because at this lowest level, NT and the RTSS are not synchronized except for the ability to atomically write the index of the last message written to the queue.

RTSS processes request NT services through this channel. Although the NT device driver interface is in the same address space of the RTSS process, the process must not call these interfaces. This restriction is necessary because the RTSS thread must wait at the message queue instead of being suspended by the NT executive.

3. RTSS Objects

Beside thread performance, the most important feature of the RTSS is the duplication of NT objects and the NT object technology. The NT object technology is described in [Custer 1993]. Adhering to the NT object technology gives Win32 fluent programmers immediate familiarity with the RTSS objects and interface.

For instance, the RTSS implements a real-time semaphore. The RTSS semaphore calls are listed in the table below with along with their Win32 equivalents.

Table 1 Semaphore Call Comparison

RTAPI Semaphore Calls	Win32 Semaphore Calls
RtCreateSemaphore	*CreateSemaphore*
RtOpenSemaphore	*OpenSemaphore*
RtReleaseSemaphore	*ReleaseSemaphore*

The RtCreateSemaphore call creates a semaphore and returns an RTSS object handle to the RTSS process. RTSS processes have a process specific object table. The RTSS object handle is an index into the object table which contains the pointers to an instance of an RTSS object.

The RTAPI calls also contain the generic object functions such as *RtCloseHandle*, *RtDupHandle*, and for objects having a signaled state, *RtWaitForSingleObject.*

The RTSS maintains other features of the NT object technology, such as name space and object retention. However, in order to keep our development effort within reasonable limits, we have not implemented resource accounting or protection. The *RtCreateXxx* calls retain the *LPSECURITY* arguments found in their Win32 counterparts so that security may be added in the future without changing the interfaces.

While leaving out the security and resource accounting simplifies the RTSS object technology, the RTSS has an additional feature not found in NT systems, which is that a class of the RTSS objects are remotely accessible from the Win32 environment and that some NT objects are remotely accessible from the RTSS environment.

Remote access to an object depends upon which of the following categories an object belongs.

3.1. Subsystem Specific Objects

In terms of remote access, a subsystem specific object is not remotely accessible by processes in the other subsystem. The prime example of this type of object is the thread and objects derived from the subsystem threads, the RTSS timer and interrupt objects. This not a serious restriction since other process' threads are not accessed except when a debugger controls a process. In the case of the two subsystems under consideration, debugging techniques are drastically different.

Denying remote access to the subsystem specific objects does not limit the usefulness of interfaces in either environment. This is the case with the interrupt handler object. The interfaces, *RtAttachInterruptVector* and *RtReleaseInterruptVector* were implemented first in the Win32 environment and later in the RTSS environment. The RTSS interrupt object gives deterministic response while the Win32 interrupt object allows easier debugging.

3.2. NT System Objects

The NT system objects are those objects to which the RTSS processes have remote access. The typical object here is the file system object. Interfaces to these objects are supported by a local RPC which passes over the communication channel described in section 2.3. The *RtCreateDirectory* call is one such example.

Because the NT endpoint is in the RTSS driver, the driver must perform the further step of translating the remote *RtCreateDirectory* into the NT device driver interfaces. As noted earlier, these calls are currently in the address space of the RTSS process, however, calling these directly will lead to scheduling disaster in the real-time subsystem.

This category of remote access easily admits equivalent functionality in the Win32 environment. For instance, the Win32 version of *RtCreateDirectory* is simply wrapper around the Win32 *CreateDirectory* call.

3.2. Shared Objects

This is the most interesting class of objects. The shared objects provide synchronization and communication between Win32 and RTSS processes. There are three objects available: the semaphore, the mailslot and the shared memory object. The semaphore and mailslot are implementations of the Win32 objects. The intent of adding a shared memory object and interface is to lessen the complexity associated with creating shared memory using the Win32 calls *CreateFile* and *MapViewOfFile*.

Win32 and RTSS applications have identical access to these objects through the RTSS driver and across the NT-RTSS communication channel. In fact, two Win32 programs can perform synchronization with an RTSS semaphore in exactly the same way as two processes would use a Win32 semaphore. This also means that a Win32 program that restricts itself to using only the RTAPI interface will synchronize without modification when it is compiled with the RTSS libraries and executed in that environment.

Using the mutex object admits unbounded priority inversion into the system when a Win32 process must run in order to release the mutex. We expect that the typical use of this class of object will be either the shared memory or the mailslot. We also expect that priority inversion by NT and starvation of NT will not be an issue in dual and multi-processor systems where at least one processor always runs NT.

4. Conclusions

To be honest, large scale modifications to the NT OS kernel were not an option since the source code is not distributed by the vender. That notwithstanding, our total effort to implement the features described is not dramatically larger for having produced a well isolated subsystem than it would have been for having made corresponding modifications to the facilities of the host OS. Furthermore, our maintenance burden going forward will be reduced dramatically.

Our implementation benefits as well from NT's consistent object semantics which provide useful guidance on how new objects or alternate implementations of existing objects should proceed. The effort to stay as compatible with Win32 as possible makes the features of the RTSS attractive to Win32 programmers and reduces the burden of dividing an application into cooperating components for host and RTSS execution environments to very acceptable levels.

4.1. Future Directions

First customer applications based on the current system have been developed and are being readied for deployment. These require manual verification of system/application timing characteristics. There is rich literature on real-time scheduling which we plan to exploit for automating the scheduling design for application components on the RTSS. Approaches where a priori processor requirements are known (deadline scheduling – [Sommer & Potter 96 and Tokuda et al. 90]) or adaptive schedule generation based on dy-namic system state such as in [Jones et al 96] are un-der consideration.

There are no provisions for process partitioning and protection in the current implementation. There is literature on safely allowing untrusted programs to execute in protected environments such as the kernel address space. How the VINO kernel protects itself is described in [Seltzer et al. 96] and an overview is given in [Small and Seltzer 96]. Another way of creating a safe environment would limit the RTSS run-time environment to a safe language such as JAVA.

References

[Custer 93] H. Custer, “Inside Windows NT.” Redmond, Washington, Microsoft Press, 1993.

[Furht et al. 91] B. Furht, D. Gorstick, D. Gluch, G. Rabbat, J. Parker and M. McRoberts, "Real-Time Unix® Systems, Design and Application Guide" Kluwer Academic Publishers, Boston 1991.

[IEEE 93] IEEE, "Portable Operating System Interface (POSIX™) Part 1: System Application Interface Amendment 1: Realtime Extension." IEEE, 1993.

[Jones et al. 96] M. Jones, J. Barrera, A. Forin, P. Leach, D. Rosu and M. Rosu, "An Overview of the Rialto Real-Time Architecture." In Proceedings of the Seventh ACM SIGOPS European Workshop, September, 1996.

[Khanna et al. 92] S. Khanna, M. Sebr\gr{e}e and J. Zolnowsky, "Realtime Scheduling in SunOS 5.0." In Proceedings of the USENIX Winter 1992 Technical Conference, January, 1992.

[MICRO 95] Microsoft, "Windows NT and Real-Time Operating Systems" Available at `http://www.microsoft.com/kb/articles/q94/2/65.htm,` 17 Jan. 1995.

[Seltzer et al. 96] M. Seltzer, Y. Endo, C. Small, K. Smith, "Dealing With Disaster: Surviving Misbehaved Kernel Extensions." In Proceedings of the USENIX Second Symposium on Operating Systems Design and Implementation, Seattle, October, 1996.

[Small & Seltzer 96] C. Small and M. Seltzer, "A Comparison of OS Extension Technologies." In Prodceedings of the USENIX 1996 Annual Technical Conference, January, 1996.

[Stodolsky et al. 93] D. Stodolsky, J. Chen and B. Bershad, "Fast Interrupt Priority Management in Operatining System Kernels." In Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures, September, 1993.

[Tokuda, et al. 90] H. Tokuda, T. Nakajima, P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System," In Proceedings of the Usenix First Mach Symposium, October, 1990.

[Sommer 96] S. Sommer "Removing Priority Inversion from an Operating System. Proceedings of the Nineteenth Australasian Computer Science Conference (ACSC'96), January, 1996.

[Sommer & Potter 96] S. Sommer and J. Potter, "An Overview of the Real-Time Dreams Extensions." In Proceedings of The Third Australasian Conference on Parallel and Real-Time Systems (PART'96), September, 1996.

[Timmerman & Monfret 96] M. Timmerman and J. Monfret, "Windows NT as Real-Time OS?" From Real-Time Magazine and reprinted at `http://www.realtime-info.be/encyc/magazine/articles/winnt/winnt.html`, 1996.

[VENTU 96] VenturCom, "High Frequency Clock and Timer Facilities." Available at `http://www.vci.com/prod_serv/nt/interim.html`, 1996.