By Dr. ir. Martin Timmerman, Managing Director,
Jean-Christophe Monfret, Project Manager,
Real-Time Consult.

Windows NT as Real-Time OS?

More and more companies are trying to use Windows NT as a standard Operating System (OS) at all levels of the industrial hierarchy. The use as server and workstation is obvious, but some people want to use it also on the factory floor. These factory floor applications demand real-time system behaviour. Can Windows NT be a component to fulfil this need?

First, we will define what a real-time system is and the OS characteristics we need to allow developers to build such a real-time system. The distinction will be made between hard and soft real-time systems. In the second part, we demonstrate how and why Windows NT cannot fulfil the requirements of a hard real-time system. We show, however, that for some simple soft real-time applications, Windows NT could be used under certain circumstances.

INTRODUCTION

Windows NT was not designed with the requirements of a Real-Time Operating System (RTOS) in mind: it has been designed as a General Purpose OS (GPOS) or, to be more precise, as a Network OS (NOS). Nevertheless, because Windows NT was created by developers of the VMS Operating System, some characteristics from the real-time world have been introduced. For example Microsoft introduced the notion of real-time class processes. They are scheduled in the same way, as it would be in an RTOS. The ISR (Interrupt Service Routine) has been designed in a very efficient way [1]. However, do these elements allow for a classification of Windows NT as a RTOS?

WHAT IS A REAL-TIME SYSTEM?

Definition

A Real-Time System responds in a timely predictable way to unpredictable external stimuli arrivals.

To fulfil this, some basic requirements are needed:

1. Meet deadlines. After an event occurred an action has to be taken within a predetermined time limit. Missing a deadline is considered a (severe) software fault.

   On the contrary, it is not considered as a software fault when a text editor reacts slowly and so enervating the user. This lack of response is catalogued as a performance problem — which can probably be solved by putting in a faster processor. It can be demonstrated that using a faster processor will not necessarily solve the problem of missing deadlines [2].
2. Simultaneity or simultaneous processing: even if more than one event happens simultaneously, all deadlines for all these events should be met. This means that a real-time system needs inherent parallelism. This is achieved by using more than one processor in the system and/or by adopting a multitask approach.

A Real-Time System responds in a timely predictable way to unpredictable external stimuli arrivals.

Hard and Soft Real-Time Systems

A classification can be made into hard and soft real-time systems based on their properties.

The properties of a hard real-time system are:

- No lateness is accepted under any circumstances
- Useless results if late
- Catastrophic failure if deadline missed
- Cost of missing deadline is infinitely high

A good example of a hard real-time system is the fly-by wire control system of an aircraft.

A soft real-time system is characterised by:

- Rising cost for lateness of results
- Acceptance of lower performance for lateness

Examples are a vending machine and a network interface subsystem. In the latter you can recover from a missed packet by using one or another network protocol asking to resend the missed packet. Of course, by doing so, you accept system performance degradation.

Other real-time systems examples are nuclear power plant control, industrial manufacturing control, medical monitoring, weapon delivery systems, space navigation and guidance, reconnaissance systems, laboratory experiments control, automobile engines control, robotics, telemetry control systems, printer controllers, anti-lock breaking, burglar alarms — the list is endless.

The difference between a hard and a soft real-time system depends on the system requirements: it is called hard if the requirement is "the system shall not miss a deadline" and soft if "the system should not miss a deadline".

There are a lot of discussions going on about the exact meaning of a hard and soft real-time system. One could even argue that a soft real-time system is not a real-time system, as the first requirement: meet deadlines, is not met. Indeed, the term "real-time" is often misused to indicate a fast system. And fast can then be seen as "should meet timing deadlines", thus meaning a soft real-time system. Therefore, we define a RTOS as an OS that can be used to build a hard real-time system.

Hard or Soft RTOS do not exist!

People often confuse the notion of real-time systems with real-time operating systems (RTOS). From time

to time people, even misuse hard and soft attributes. They say this RTOS is a hard RTOS or this one is a soft one. There is no hard RTOS or soft RTOS. A specific RTOS can only allow you to develop a hard real-time system. But having such an RTOS will not prevent you from developing a system that does not meet deadlines.

If, for example, you decide to build a real-time system that should respond to an Ethernet TCP/IP connection, it will never be a hard real-time system as the Ethernet itself is never predictable.

Of course, if you decide to build an application on top of an OS like "Windows 3.11", your system will never be a hard real-time system as the behaviour of the "OS" software is by no means predictable.

NECESSARY OS REQUIREMENTS FOR AN RTOS

Req. 1: an RTOS has to be multi-threaded and preemptible.

As mentioned above, a RTOS should be predictable. This does not mean that a RTOS should be fast, but that the maximum time to do something should be known in advance and should be compatible with the application requirements. Windows 3.11 — even on a Pentium Pro 200 MHz — is useless for a Real-Time System, as one application can keep the control forever and block the rest of the system (Windows 3.11 is co-operative).

We define a RTOS as an OS that can be used to build a hard real-time system.

The first requirement is that the OS is multi-threaded and preemptible. To achieve this the scheduler should be able to preempt any thread in the system and give the resource to the thread that needs it most. The OS (and the hardware architecture) should also allow multiple levels of interrupts to enable pre-emption at the interrupt level.

Req. 2: the notion of thread priority has to exist.

The problem is to find which thread needs a resource the most. In an ideal situation, a RTOS gives the resources to the thread or driver that has the closest deadline to meet (we call this a deadline driven OS - see also [2]). To do so, however, the OS has to know when a thread has to finish its job and how much time each thread needs in order to do so. For the time being there is no RTOS for which this is the case, as it is so far too difficult to implement.

Therefore, the OS developers took another point of view: they introduced the concept of priority levels for threads.

The designer is responsible for converting deadline requirements in thread priorities. As this human activity is prone to error, a Real-Time System can easily go amiss. The designer can get help in this transformation process by using for example Rate Monotonic Scheduling theories [2] and some simulation software [9] but it can be of no avail. Nevertheless, as there is no other solution today, the notion of thread priority has to exist.

Req. 3: the OS has to support predictable thread synchronisation mechanisms

As threads share data (resources) and need to communicate, it is logical that locking and inter-thread communication mechanisms exist.

Req. 4: a system of priority inheritance has to exist.

In fact, it is these synchronisation mechanisms and the fact that the different threads run in the same memory space that makes the difference between threads and processes. Processes do not share the same memory space. For instance, old UNIX versions are not multi-threaded. (Old) UNIX is a multitask OS, but the tasks are processes that can only communicate via pipes and shared memory. Both mechanisms use the file-system characterised by an unpredictable behaviour.

The combination of thread priority and resource sharing between them leads to another point: the classical problem of priority inversion. Before having a priority inversion condition, at least three threads have to be involved. When the lowest priority thread has locked a resource (shared with the highest one) while the middle priority thread is running, the highest priority thread is then suspended until the locked resource is released and the middle priority thread stops running. In such a case the time needed to complete a high priority level thread depends on a lower priority level "priority inversion". It is clear that in such a situation it is hard to meet deadlines. (Figure 1).

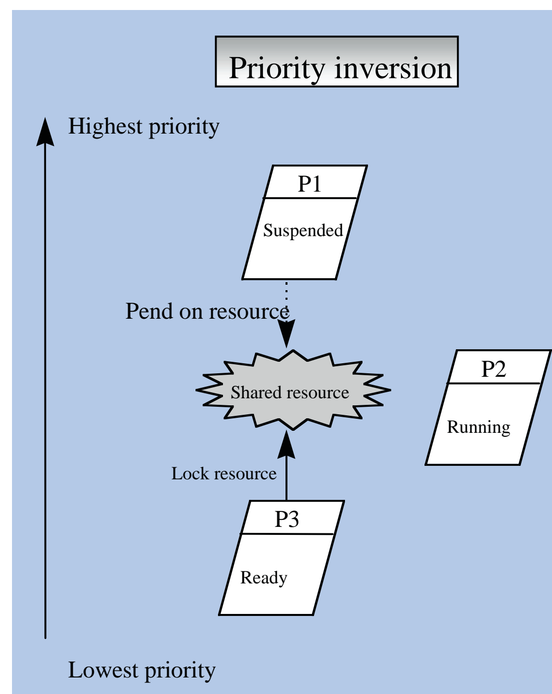

Figure 1. Priority inversion

To avoid this, a RTOS should allow priority inheritance in order to boost the lowest priority thread above the middle one up to the requester. Priority inheritance means that the blocking thread inherits the priority of the thread it blocks (of course, this is only the case if the blocked thread has a higher priority).

Some people will argue that a correct design would avoid the problem. This is not true for complex applications. The only way to solve this is to increase the priority of the thread manually before locking a resource. This implies of course that only two threads of different priorities are sharing the protected resource. If not, there is no solution.

The notion of thread priority has to exist.

OS Behaviour should be known

Lastly, the timing requirements should be considered. In fact, the developer should know all the timing of system calls and the system behaviour in all his flavours. Therefore, the following figures should be clearly given by the RTOS manufacturer:

- The interrupt latency (i.e. time from interrupt to task run): this has to be compatible with application requirements and has to be predictable. This value depends on the number of simultaneous pending interrupts.
- For every system call, the maximum time it takes. It should be predictable and independent from the number of objects in the system;
- The maximum times the OS and drivers mask the interrupts.

The developer should also know the following points:

- System Interrupt Levels.
- Device driver IRQ Levels, maximum time they take, etc.

When all the previous metrics are known, one can imagine to develop a hard Real-Time System on top of this OS. Of course, the performance requirements of the System to develop should be compatible with the RTOS and the hardware chosen.

DOES WINDOWS NT FULFIL THESE REQUIREMENTS?

Windows NT is multi-threaded and preemptible

It is clear that Windows NT is a multi-threaded and preemptible OS [3], as such it fulfils the first requirement (Req. 1). Let us now examine the predictability of Windows NT. Therefore, we have to recall the structure of Windows NT and to figure out if and how real-time aspects of a system can be handled in Windows NT.

Thread priorities (Req. 2)

In a real-time system, not all the jobs have the same priority. Some of them (those that are time critical) have higher priorities. Others like displaying the state of the system, logging events to file or configuring the system have on the contrary lower priorities. To take this into account, the OS should be able to assign different priorities for these jobs.

In Windows NT the notion of priorities is quite complex:

- There are two classes of process priorities: real-time class and dynamic class. Processes in the real-time class have a fixed priority level that can only be changed by the application itself, whereas the processes in the dynamic class will see their priorities changed by the scheduler depending on the type of work the process is doing (interactive or non-interactive activity). Only the first class can be used in a real-time system to guarantee the predictability of the system. It should be noted that for a non real-time job of a real-time system the second class could always be used as long as it does not share resources with the process of the real-time class.
- A process has a base priority level that can only be changed by the application itself (in the case of the real-time class).
- A thread in a process can have a priority in the range of ±2 the process base priority level plus the two extreme levels of the class. For example the threads of a process of base priority 24 (Real-Time Class) can have any priority level between 22 and 26 plus the levels 16 and 31.

Figure 2 represents these two classes of processes.

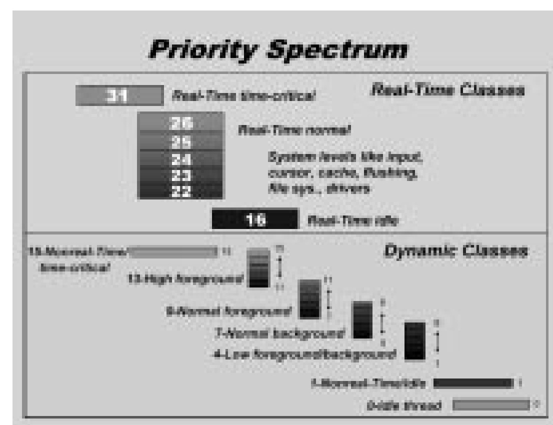

Figure 2 Windows NT priority spectrum

As there is a notion of thread priority, the second requirement is met. But there is a problem: only a small number of priority levels are allowed in Windows NT. Most of the modern RTOS allow for at least 256 priorities.

Why is this a problem? The answer is obvious: the more priorities you have, the more predictable your system is predictable. The best way to design a system is to assign to each different thread a different priority [2]. For a given process in Windows NT you have only 5 (7 if you count the two extreme ones) priorities for threads in a given process. So doing, a lot of them will have to run at the same level. Your predictability will then be mediocre if you have to handle more than one or two critical events. Some would say that different processes could be. But even then, the

total number of priorities is only 16. Moreover, context-switching time between threads from different processes is much higher than between threads in the same process, as they do not share their memory space, which is a part of the context. Therefore, this is not a good solution.

Priority inversion

As mentioned previously the problem of priority inversion is a classical one in a real-time system. Therefore, in a RTOS the synchronisation system calls such as mutexes, semaphores and critical sections should be capable of handling priority inheritance. In Windows NT this is NOT POSSIBLE [4], so the requirement 4 is not met.

Another problem related to the priority inversion in Windows NT is due to the implementation of some GUI system calls. These are handled in a synchronous way (the calling thread is suspended until the completion of the system call) by a process running at a non real-time class priority [5]. So doing, a lower real-time class thread can prevent the upper ones from running.

As we have seen, the relatively low number of priorities correlated to the problem of the priority inversion make windows NT only suitable for simple (few different kinds of events) Real-Time Systems.

Win32 API characteristics

Why then choose Windows NT? As mentioned before, it could be interesting to have the same OS at different levels in the company's industrial hierarchy. But another very interesting point to consider is the very rich and powerful Win32 API. You have a lot of good development platforms, good compilers and also a lot of engineers that know this API well. The features of this API are numerous. There is everything you need to create a powerful multi-threaded application. The only question is to know if you can build a reliable Real-Time Application on top of it.

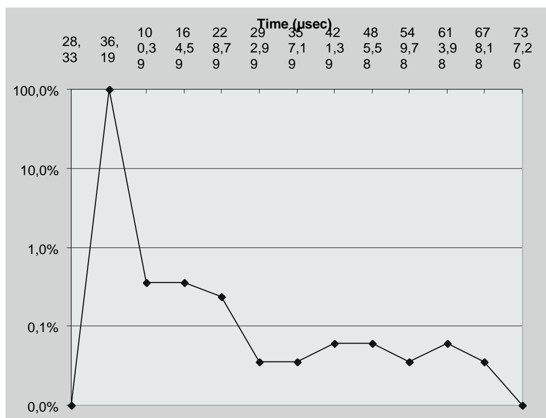

Figure 3 Get Mutex System call time

To do so, the system call should be predictable and independent from the number of objects in the system.

To measure this we have done a few simple tests. We have designed a process (belonging to the real-time class) that has a thread executing synchronisation system calls. We used the QueryPerformanceCounter system call before and after each synchronisation system call to measure the time it takes. At the end of the test, we saved everything on disk to exploit the results with Excel (Figure 3).

We made sure that no swapping had occurred during the tests and that no resource had been locked -so our measures are significant. We have also measured the delay introduced by the tracing activity and removed it from the test results. The tests have been performed on a 100 MHz Pentium with 24MB RAM.

For the request mutex system call, for example, we got an average of 35 µs. But the maximum has once been up to 670 µs!

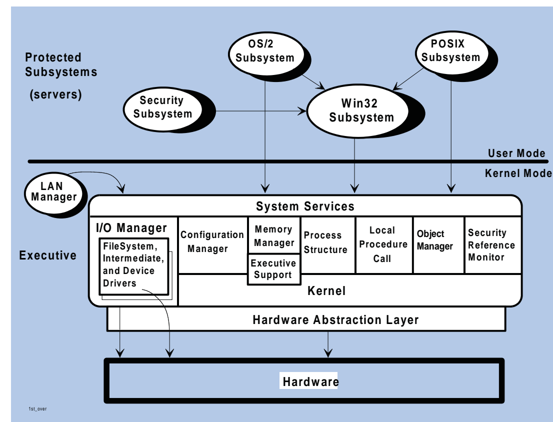

Figure 4 Windows NT hardware interfacing

Why? This is simply due to an interrupt either from the disk or from the network as this was the only possible activity of the system (this machine is connected on our network). This was reproduced many times when loading artificially the system with disk and network transfers while performing the tests. So doing, we ended up delaying the system calls up to a few milliseconds!

The Win32 API as we said is very rich. But it should be noticed that it is not well designed for real-time. For example mutex requests are queued in FIFO order and not in priority order [10]. This is a big drawback for predictability. Great care should also be taken when choosing a system call. For instance, for thread synchronisation in the same process, the critical section should be preferred to any other call (this call takes only a few microseconds compared to 35 µs with Mutex). This call is designed to work only in such a case (threads have to belong to the same process). This is normal, as the context switch between two threads of the same process is much lighter than the one between two processes.

A final remark concerning this API: when using an API system call one should notice that some of them are executed by system processes running at a lower priority (dynamic class) and are synchronous. So doing, the calling thread will wait for the completion of the call. This could result in a priority inversion.

Our conclusion after these very simple tests is that, despite a powerful API, the underlying kernel and I/O manager is not well adapted to handle real-time

events at the application level!

We will discuss later if this problem can be solved at the driver level.

Interrupt Management

A real-time system interacts with the external world via the computer hardware. External events are converted into interrupts and handled by a device driver.

Figure 4 from the DDK Documentation [6], shows how Windows NT is designed to deal with the hardware.

The only way to reach the hardware is via a kernel device driver. This is logical in such an OS as processes are running in separate memory spaces and cannot gain access directly to the hardware. As most real-time applications are dealing with special external events, the developer must also develop the kernel device drivers to gain access to the hardware. Let us see how a device driver is designed.

Interrupt	Definition
Level 31	Hardware error interrupt
Level 30	Powerfail interrupt
Level 29	Inter-processor interrupt
Level 28	Clock interrupt
Levels 12-27	These levels map to the traditional interrupt levels 0-15 used in PCs
Levels 4-11	These levels are not generally used
Level 3	Software debugger interrupt
Levels 0-2	Reserved for software-only interrupts to prioritize work within device drivers and executive components

Figure 5 Windows-NT Interrupt levels

The device driver is responsible for handling the interrupt generated by the hardware it controls. To do this, an original mechanism has been chosen in order to increase the responsiveness of the OS. The interrupts are handled in two stages. First, the interrupt is handled by a very short Interrupt Servicing Routine (ISR). Afterwards the work is completed by executing a DPC (Deferred Procedure Call). This delivers the following event flow:

- Interrupt occurs
- Processor saves PC, SP and calls the dispatcher
- The OS saves the context and calls the ISR
- The device driver handles the ISR in the shortest possible time, doing only the critical work (Reading/Writing hardware registers)
- The device driver calls a Deferred Procedure Call function (in fact this is a queue handled by the I/O Manager)
- The device driver exits the ISR
- The OS restores the context
- The processor restores the PC and SP
- The pending DPC is then scheduled at DISPATCH-LEVEL priority. It is a kind of software interrupt.
- After completion of all pending DPC, the OS reschedules the user applications.

As we can see the handling of an interrupt is quite complex in Windows NT. Moreover, an ISR must be designed in the correct way to be as short as possible. So most drivers do a lot of work in the DPC (which are only preemptible by an ISR) causing other DPC to wait for their completion as all DPC run at the same priority level (DISPATCH_LEVEL, level 2 on Figure 5).

The Windows NT Device Driver Documentation tells you that: "the ISR are preemptible by upper priority ISR and that DPC have higher priority than user and system threads". But as all DPC are running at the same level and as the policy that should be used when designing a Device Driver is to reduce the ISR to the minimum (deferring the rest of the critical job in the DPC), your DPC will have to wait for other DPC to be completed. Therefore, here your application will depend on others device drivers.

Is this different in a RTOS? Of course it is. In an RTOS the developer first knows at which level the other device drivers are running. There is usually some free space for interrupts above the standard drivers. All the critical work is then done in the ISR allowing the developer to tune its drivers' configuration depending on the time constraint of the application. In Windows NT the ISR are very fast, so the interrupts are not blocked for a long period, but nothing is done in the ISR. Most of the job is deferred to the DPC. So if you are using a third party device driver badly designed, you will finally miss deadlines, except if you do not follow Microsoft policy and do the entire job in the ISR — but then, why use an OS?

Notice here that this could be very crucial as we discovered that some DPC from hard disk and network drivers take more than a few milliseconds.

Memory Management

Another point to consider when designing a real-time system is the memory management policy of the RTOS. In Windows NT, processes are in their own me-mory space. Virtual memory mechanism and a pa-ging system can only achieve this. For business applications, this is great, but for a real-time system

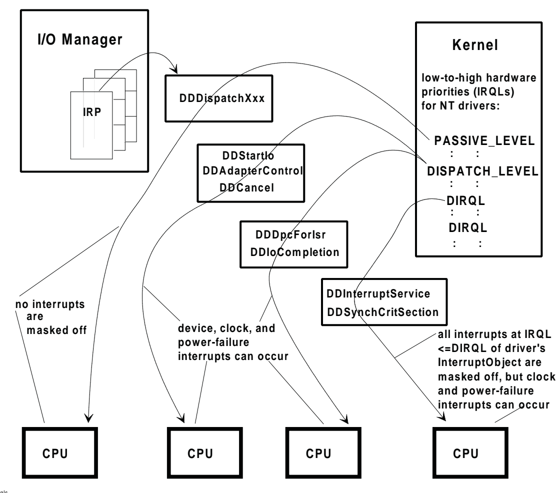

Figure 6 Pre-emption at the interrupt level

Microtec Research Ad

that has to respond to external events in a predefined time limit, this generates unpredictability when the system has to get a memory page from disk. Therefore, Windows NT allows you to lock pages into memory through the call to the VirtualLock function. . In his book, Jeffrey Richter says that Windows NT can still unlock the pages and swap them out to the disk if the whole process is inactive [7]. However, we have not manage to demonstrate this. The only thing we manage to find that the memory allocated for the window can still be swapped out when you iconise them. So normally for Real-Time applications, this will not be a problem. Is the previous point still a problem? It is not, if the application and other third party applications are well designed and do not take more memory than physically available.

At the device driver level, however, swapping can be avoided.

Other points to consider

Some other few points should be considered when trying to use an OS for real-time or embedded applications:

- For embedded applications, the memory footprint is generally a key issue. Windows NT has quite a big memory footprint compared to other RTOS on the market.
- Most of the time you do not need any keyboard or display.
- For large series productions, the Windows NT license is really too expensive: $ 255 for NT 4.0 Workstation (You can reach $ 186 for NT3.5 with more than 3000 units and even less, for very large quantities!)

Commercial Side

Last but not least, one should consider the fact that some companies are offering solutions to make Windows NT more Real-Time — a proof if any that it is not the case by default! Likewise, this shows that there is a market for it.

CONCLUSION: COULD WINDOWS NT BE USED AS A RTOS?

We showed in this article that Windows NT, as it has been designed to deal mainly with classical applications, is not a good platform to support Real-Time applications:

- the number of available priorities in the real-time class is too low for real-time applications;
- the problem of priority inversion is not solved in the OS (for the real-time class process);
- for embedded applications, the memory footprint is too big and the license too expensive;
- device drivers can take a lot of time in DPC and no pre-emption by other DPC is possible.

So why do some people say they can manage to use Windows NT for real-time applications [8]? First, one specific application never demonstrates anything on its own. Second, Windows NT can only be used under the following circumstances:

- Soft Real-Time Systems whose requirements allow for deadlines to be missed from time to time;
- Simple system where the number of different kinds of events is not too big (predictability of DPC is then higher);
- CPU load always remains low (System DPC have time to execute);
- Low number of/or no third party drivers or at least a good choice of third party drivers. (source code needed to check!);
- Critical jobs (or even all the jobs) done at the DPC level or better in the ISR itself.

This last point, however, makes the use of any OS pointless.

But, for hard Real-Time systems, Windows NT is out of the question. Your system will never be predictable enough. To put it bluntly — you can rely upon it and take a plane: Windows NT will not be used in any fly-by-wire systems in the near future!

But in earnest, what should be changed in Windows NT to enable its use in hard real-time systems? The real-time class process should have more priority levels. The problem of the priority inversion should be solved. The whole interrupt system should be changed:

- having DPC is a great idea but they should have many priority levels;
- DPC should be preemptible by other higher priority DPC;
- third party drivers and system drivers should be configurable (ISR priority level, DPC priority level);
- third party drivers should be detailed to the developer, who should at least know the maximum time a device driver can take (during ISR, during DPC);
- the time the OS masks the interrupt should also be known to the developer;
- the time each system call takes should finally be specified to the developer.

Is Microsoft willing and ready to introduce such enhancements in Windows NT or do they find the market to small to let this open for third parties? ■

Dr. Ir. Martin Timmerman is graduated in Telecommunications Engineering at the Royal Military Academy (RMA) Brussels and is Doctor in Applied Science at the Gent State University (1982). He became the director of the System Development Centre (SDC) at RMA, which he created in 1983, and converted himself to a Computer Science Engineer. Actually he is giving general courses on Computer Platforms and more specific courses on System Development Methodologies at RMA. Outside RMA, Martin is known for his audits, reviews and seminars for his two companies Real-Time Consult and Real-Time User's Support International. RTUSI provides hardware and software support services and is involved in project engineering for Real-Time Systems.

Ir. Jean-Christophe Monfret graduated in 1993 as an Ingénieur Télécom Bretagne, France. He has a

master's degree in Parallel Computer Science from the University of Rennes I, France (1993), a specialisation he had the opportunity to put to practice for a year at the Commissariat à l'Energie Atomique, France. He has been working since 1994 for RTUSI and Real-Time Consult as a project manager where he is also involved in audits and review activities.

REFERENCES

1. Technology Brief: Real-Time Systems with Microsoft Windows NT, Publication of the Microsoft Developer Network, Microsoft Corporation, 1995.
2. J. Zalewski, "What Every Egineer Needs To Know About Rate-Monotonic Scheduling: A Tutorial," in Real-Time Magazine, Issue 1995/1 (http://www.realtime-info.be)
3. Joel Powell, Multitask Windows NT, Waite Group Press, 1993.
4. Microsoft® Win32® Software Development Kit Documentation.
5. Helen Custer, Inside Windows NT , Microsoft Press, 1992.
6. Microsoft® Windows NT™ Version 3.51 Device Driver Kit Documentation.
7. Jeffrey Richter, Advanced Windows, The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95, Microsoft Press, 1995.
8. Brian Catlin, Design of a TCP/IP Router Using Windows NT, a paper presented at the 1995 Digital Communications Design Conference, Catlin & Associates, Redondo Beach, CA.
9. P. KORTMANN and G. MENDAL, PERTS Prototyping Environment for Real-Time Systems, Real-Time Magazine, Issue 1996/1 (ISSN 1018-0303).
10. Mutex Wait is FIFO but Can Be Interrupted, in Win32 SDK Books Online, Microsoft Corporation, 1995.

Editorial schedule Real-Time Magazine

Quarter	Issue	Topic
1Q 97	Buses	Overview of buses used in real-time systems. Including: "The right bus on the right place, Part II." **VME** - Ariel Corp., CETIA, LSI, Sky Computers. **CompactPCI** - CES, Hybricon Corp., I-Bus, MEN Mikro Elektronik, Teknor Industrial Computer, Interphase Corp. **Mezzanines** - GRoupIPC, MUMM. **RACEway** - Cypress Corp., Myriad Logic
2Q 97	RTOS Update I	Overview of (Unix-like) RTOS. Including special report on **Windows-NT** and its real-time extensions: QNX Software Systems, Radisys Corp., VenturCom, LP Elektronik, Accelerated Technology. **POSIX** - Lynx RTS, Modcomp. **CHORUS** - Chorus Systems. **DSP** - Eonic Systems, Spectron Microsystems. **HW SUPPORT** - Digital Equipment Corp. + **RTOS Vendor List**
3Q 97	RTOS Update II	Overview of (non-Unix like) RTOS. Including articles from Embedded System Products, Enea Data, Eyring Corp., Express Logic, Green Hills Software, Integrated Systems, Microprocess Ingenierie, Microprocessing Technologies, Microtec Research, Performance Technologies, TRON Association, University of Karlsruhe, Wind River Systems. + **VRTX Evaluation** + **RTOS Buyer Guide.**
4Q 97	Industrial PC & PLC and their networks	In industrial process control and other fields, PLC's are a lot used. During the last years, we have seen that more and more people are tempted to use PC's for control and measurement applications. This magazine will clarify the state of the art in this field + **first Board Evaluation.**

For more information visit Real-Time Magazine online at http://www.realtime-magazine.com
To contribute your paper, contact us at 32-2-520.55.77 or e-mail to info@realtime-magazine.com