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

Windows NT Real-Time Extensions: an Overview

Windows NT (NT) becomes increasingly popular to be used in a Real-Time System. There are several reasons for that:

- ***The Win32 API is considered as a standard and a tremendous number of software is available for it.***
- ***The graphical user interface is so popular that only a few competitors are left now.***
- ***NT has a lot of ready to go solutions for communication issues.***
- ***All sorts of development tools are available for Window-NT environment.***

In a previous article [1] we discussed the impossibility of using NT as a Real-Time OS. (RTOS). In this article, we want to discuss and compare the solutions the industry is offering today to give NT a Real-Time flavour.

All solutions have potential advantages and drawbacks. This essentially means that each of them is probably useful in only a particular application type. The information in this paper should give the reader the capability of choosing the right solution for his kind of application.

INTRODUCTION

There are different approaches to use NT technology inside your Real-Time System:

- use NT as it is in a soft real-time application
- implement a Win32 API on top of another RTOS
- make coexist on one processor NT and a RTOS (or part of it)
- use an architecture with more than one processor where NT runs on one (or more) and the real-time part on another (or more).

However, before dealing with these solutions, we should first discuss how a real-time system could be constructed

HOW TO SATISFY REAL-TIME REQUIREMENTS?

Introduction

Our article on Window NT as Real-Time OS has been available on the web since 3 months now. We have received many reactions. They can be subdivided in 2 categories:

- positive reactions from users who are happy to see an independent analysis
- negative reactions of some manufacturers who want to sell real-time NT solutions to their (not always informed) customers.

In this discussions it has become clear that a lot of vendors have their own idea of what real-time means and how you satisfy real-time requirements. Some are only thinking in terms of interrupt routine handling, others are considering the use of a multitasking concept to deal with the problem. I think it is therefore necessary to give here an overview of how we see you can satisfy real-time requirements.

It has become clear that a lot of vendors have their own idea of what real-time means and how you satisfy real-time requirements.

Figure 1 gives a global overview of how RT Requirements can be implemented. This figure gives no absolute values. It gives an indication of how the spectrum RT-System spectrum is subdivided. Indeed, the x-axis speaks about deadline. There is of course a great difference if you have to satisfy one or 10 deadlines simultaneously. The speed of the processor has also a tremen-

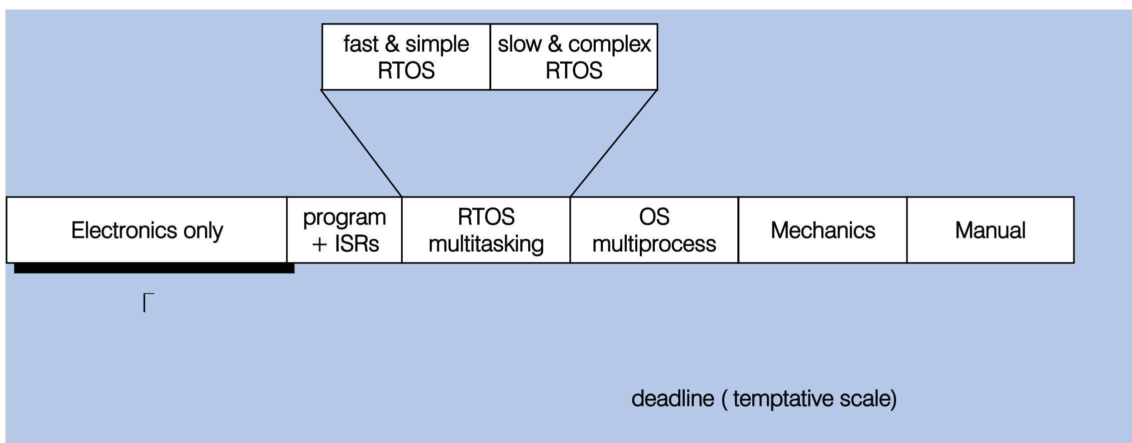

Figure 1. RT-System Classification

Enea Data Ad

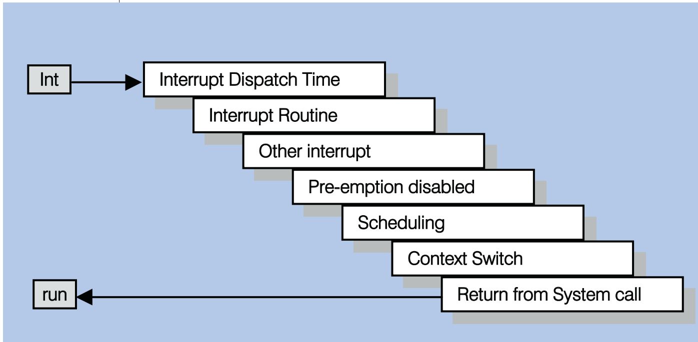

Figure 2. Interrupt to task run

dous influence on the figure. If you have a 10 times faster processor, you may shift one time decade. If you need exact figures, you should apply theories like Rate Monotonic Schedu-ling [2] or others.

In this article we discuss the use of an OS in to satisfy the RT requirement. The hardware only and processor without OS are excluded in this discussion.

Deadline response in a thread or task

It is now important to see how we can respond to an event and how we can meet the real-time requirement of reacting within a prescribed time limit to that event (making the system predictable).

The most classical way of dealing with this problem, using an OS, is to detect the event via an interrupt and to start a task or a thread to handle the work to be done. There is a latency between the occurrence of the event and the task start, let us call this the Task Latency (TL). This is not the Interrupt Latency (IL), which is the time between the occurrence of the event and the start of the Interrupt Service Routine (ISR). This ISR is in most cases part of the device driver.

It may take some time before a thread or task is started after the interrupt arrived and as we discuss real-time systems, the worst case should be considered. 7 different activities have to happen before the task runs (not necessary in this order): (figure 2)

1. interrupt dispatch time: the time needed by the hardware to get the event converted to an interrupt + the time needed to get that interrupt to the processor
2. interrupt routine: the execution time of the code in the interrupt routine
3. other interrupt: the time needed by the OS to keep track of other incoming interrupts
4. pre-emption disabled: the time needed by the OS to execute some critical code where no thread or task may be executed
5. scheduling: the time needed to decide which task is going to run
6. context switch: the time needed to switch from the previous context to the new (task) one
7. return from system call: the extra time needed to get things organised if the interrupt occurred during the execution of a system call.

This delays have been published by Lynx for LynxOS.

Worst case is 210 microseconds if 3 more interrupting devices are simultaneously there. The interrupt dispatch time depends on the hardware around the processor. The interrupt routine execution time depends on the speed of the processor and the number of lines of code to execute. The time lost for other interrupts depends on the mechanism in the OS and the hardware to deal with these simultaneous interrupts. The pre-emption disabling time has to do with the complexity of the OS, the number of lines of code to execute which are critical compared to the threads (or tasks). Context switch time depends on the definition of the context and the speed of the processor. The time lost if the interrupt occurs during a system call depends on the complexity of the processor and his vector handling.

Similar figures have been published for the HP-UX implementation of LynxOS on a PAT processor. These show clearly the influence of the processor interrupting mechanics on the times (in the good sense compared to a 386). But it shows also that the quality of a port of the OS will influence a lot too (pre-emption disabling time is very long for this complex processor compared to a simple 386).

Superconducting Supercollider Laboratory (SCC) has executed in 1993 a benchmark program and they have been measuring this particular figure we have been discussing here for different OS. Some manufacturers were upset about the figures and published new ones taking into account the recent versions of their OS. The last publication dates from an article from J. Ready. [4] These figures have not been mea-

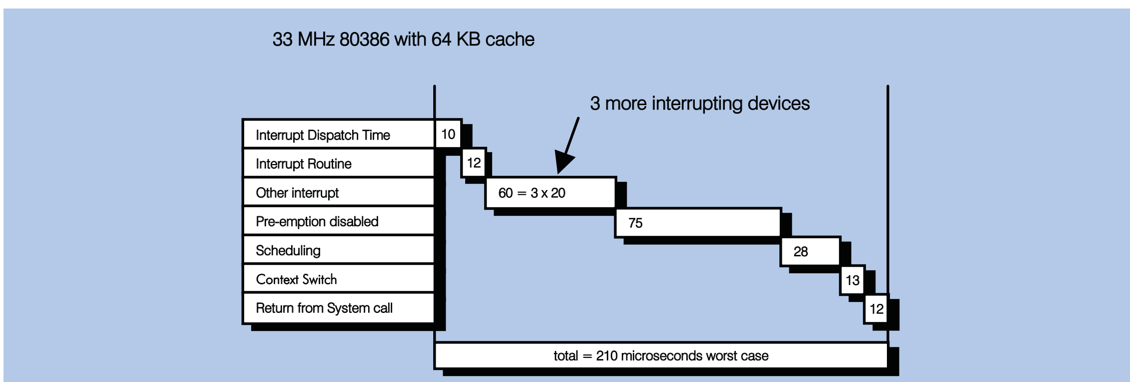

Figure 3. LynxOS Performance metrics - interrupt to task run

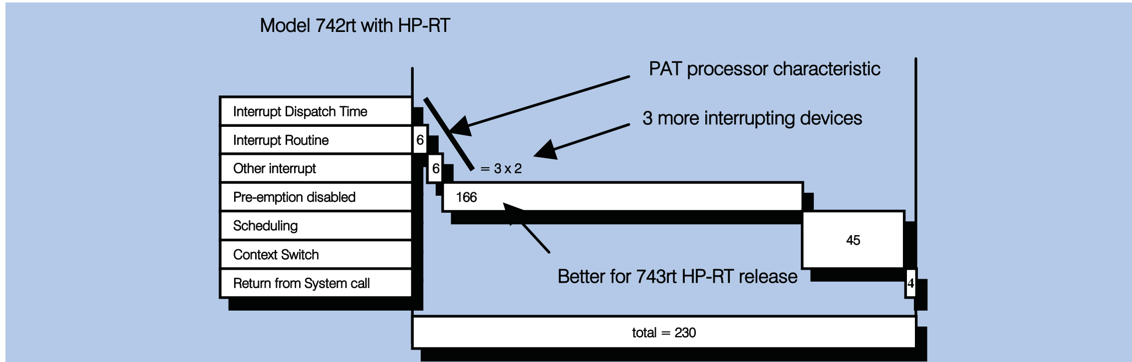

Figure 4. HP-RT Performance metrics - interrupt to task run

sured by an independent party so we cannot guarantee their correctness, but this is not the topic here. We are only interested in the absolute value of these figures to understand how much time is lost (in worst case) between an interrupt and a task run. (Figure 5)

They test all this on a 68030 processor on the very popular VMEbus board MVME147 from MOTOROLA. The maximum response time in the figure is important because this is the only one we can take into account if we talk about hard real-time requirements. (hard real-time means: deadlines may not be missed, soft real-time means: deadlines may be missed sometimes. This corresponds then to a performance degradation of the system).

A fundamental conclusion of these figures is that we don't have to expect to handle events within microseconds if we use an OS. We see that we need approximately 100 up to 250 microseconds before we start the handling of the event in the thread or task. Then we have still to count for the thread execution time before we arrive to the deadline. If you have for example a deadline within 1 millisecond, then you have 750/900 microseconds execution time left. It will then largely depend on the MIPS rate of your processor how complex your handling may be.

If you make an application using this technique, you will end up in an acceptable clean, portable application because most of the treatment you have to do is at thread level. A standardised API towards the OS is then very important. One solution is to go for POSIX compliance, the other is to go for the WIN32 interface, but this story will be discussed later.

However, for a lot of applications these TL are too long and therefore some other techniques exist to deal with the handling of a real-time events.

Deadline response on other levels

Fundamentally, you can imagine dealing with an event on different levels in the OS. Depending on the features of the OS, up to 4 levels can be discovered in literature. To be general, let us limit ourselves here to 3 levels: interrupt service routine, device drive level and thread level.

1. ISR:

   one can imagine to put all the code to be executed for the event in the ISR. At that moment, you are totally independent of the OS. In a way you deal with the problem as if you had only a program & ISR's from figure 1. Here very short handling times are involved. Only the interrupt dispatch time, the other interrupt and the pre-emption disabled (as a RTOS can mask interrupts) times are involved. We are so talking about 10 and less microseconds. However, the length of the ISR may have a considerable negative effect on other interrupts to be handled and in general on the total system behaviour. This method is therefore only advisable if only one or two events are to be handled in the 50-microsecond range. This means approximately 40 microseconds execution time for the event. Some vendors of NT-RT technology are using this method and are claiming indeed these 50 microseconds response times. Don't forget that you are using no other interface than the processor and that this code is not portable at all. Debugging this part of the system may be very hard and specialised work not open to all of us.

2. Device Driver:

   A mid between solution is to deal with the event on a device driver level. Then you can make short interrupt routines and postpone the handling of the event to the background (sometimes called

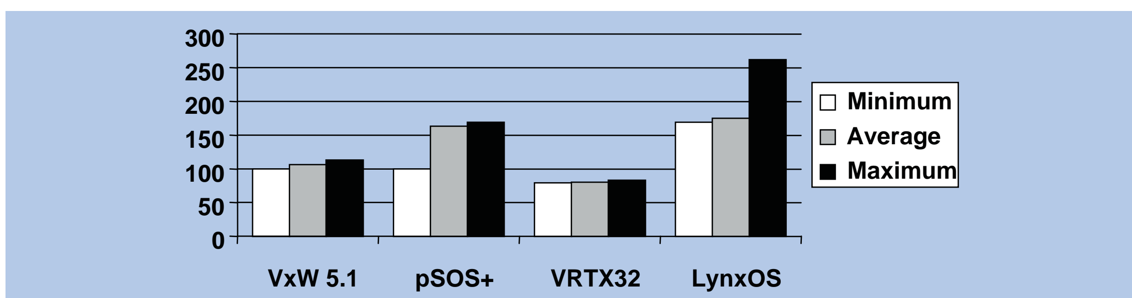

Figure 5. MVME147 Interrupt to task response times [4]

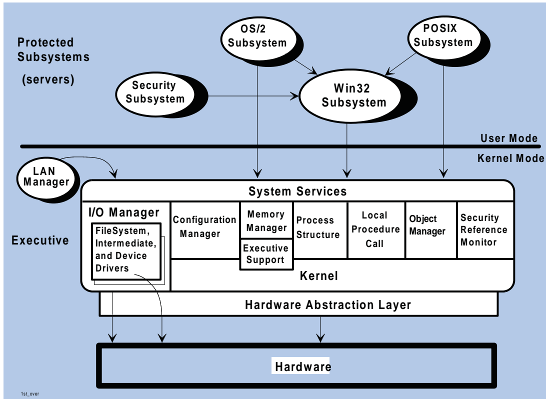

Figure 6. Windows NT hardware interfacing

level 0 interrupt). The code executes on the level of the OS. The reaction times you can deal with here are somewhere between the 50 and 250 microseconds. Again, it may be hard to debug as you are debugging parts of the OS. You have no standard interface (= no portable code) and the knowledge you need about how the OS mechanics are working on device driver level are much greater than in the previous case.

3. Thread:

   this is the normal approach, which we discussed to begin with.

In this discussion, we have looked toward the latencies. Once you have a predictable system (there are and you know the maximum latencies), then the problem arises of how to deal and stay predictable if you start dealing with more than one event simultaneously. This largely depends on the available processor power. Techniques like RMS [2,3] can be applied to verify if the system can meet all the deadlines. This is beyond the scope of this article, but the reader should be aware of the fact that having predictable latencies is no guarantee for a predictable system.

HOW DOES WINDOWS NT DEALS WITH EXTERNAL EVENTS?

Introduction

Some NT-RT technology vendors apply one of these techniques to sell you "very fast hard real-time solutions". Is this possible? We are eager to study the different solutions presented in the market place, but before we can do that, we need a closer look on how NT deals with interrupts and device drivers.

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

There are two ways to look towards this mechanism: from the application to the hardware and from the hardware to the application.

The only way to reach the hardware from an application is via a kernel device driver. For NT this is logical, as in such an OS processes are running in separate memory spaces and cannot gain access directly to the hardware.

As most real-time applications are dealing with one or more special external events, the developer must also develop the kernel device drivers to gain access to the hardware for these special events. Let us see how this works in NT.

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). Figure 7 explains how this works. Three levels are involved: interrupt level, dispatch level and user level. The DPC runs at the midbetween dispatch level in kernel mode.

This mechanism needs an ISR to be designed as short as possible. Most drivers do a lot of work in the DPC (which can only be pre-emptied by an ISR). The dispatch level corresponds with only one priority level (DISPATCH_LEVEL, level 2).

The Windows NT Device Driver Documentation tells you that: "the ISR are pre-emptible by upper priority ISR and that DPC have higher priority than user and system threads". It also tells you that:

- all DPC are running at the same level
- the policy that should be used when designing a

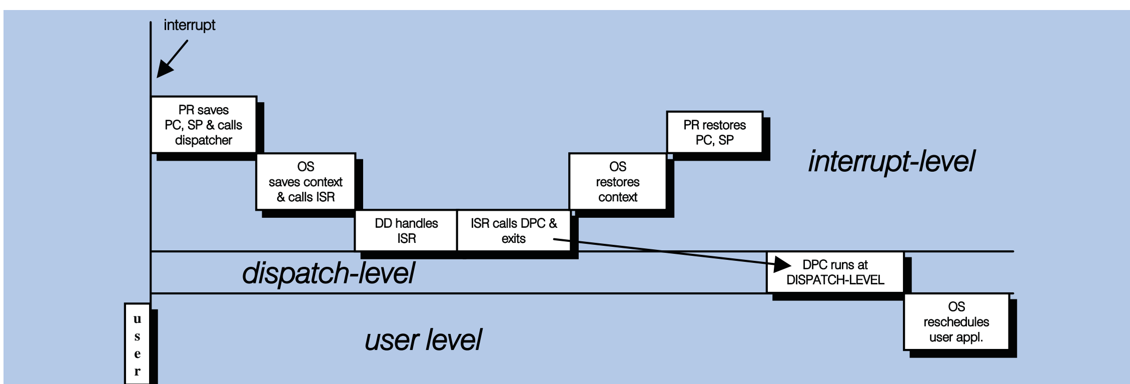

Figure 7. NT interrupt handling flow

Greenspring Ad

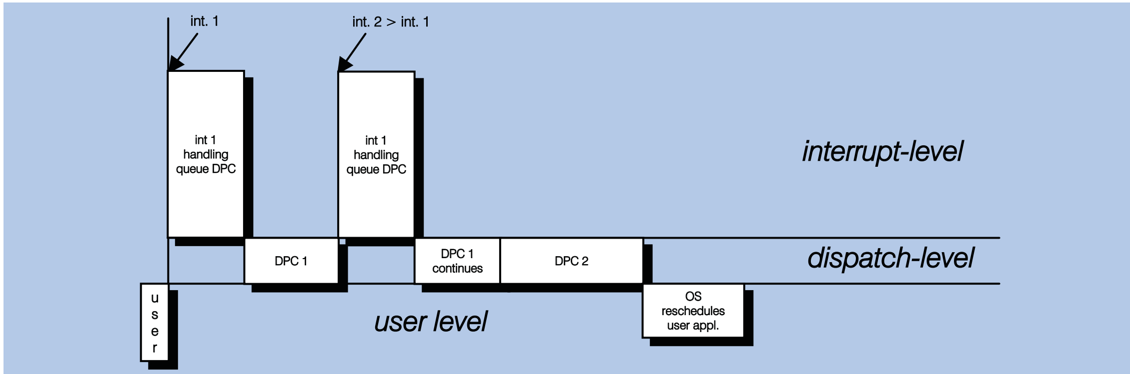

Figure 8. Dispatch level FIFO

Device Driver is to reduce the ISR to the minimum (deferring the rest of the critical job in the DPC)

- the DPC are queued at this level in a FIFO form. One DPC has therefore to wait for the execution of the previous started DPC. This may also end up in priority inversion in the DSP queue.

Therefor we can conclude that your DPC will have to wait (long) for other DPC to be completed, and thus your device driver response time will depend on others device drivers.

Some people react on our online article saying that it is possible to queue a DPC in front of the pending DPC queue. But mixing FIFO and LIFO mechanisms in the same queue, could make thins even worse (except if you have only one urgent event to deal with).

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.

This shows clearly that you cannot rely on NT as it is now for RT applications. NT can only be used in the case where you have to deal with a very limited number of events and you deal with them in the ISR. But this solution is not producing a clean (portable) application architecture.

COMMERCIAL SOLUTIONS

The different solutions

There are different approaches to use NT technology inside your Real-Time System:

- use NT as it is
- implement a Win32 API on top of a RTOS
- make coexist on one processor NT and a RTOS (or part of it)
- use an architecture with more than one processor where NT runs on one (or more)

use NT as it is

You can first imagine to use NT as it is including the interrupt handling structure as explained in the previous paragraph. Remember that NT has not been designed to be a RTOS, so take care of what you are doing. Figure 9 represents in a simplified way the structure of NT.

Some people enhance the interrupt handling mechanism. The only way of intercepting the mechanism is to intercept the interrupts. Therefore you need to add a special hardware extension. One vendor (LP-Elektronik) for example intercepts interrupts and uses then the NMI (not used at NT level) to deal with the real-time events. This is only applicable if the processor has a separate interrupt stack. The NMI routine should be written with care: no OS call should be issued, and be as short as possible not to loose other interrupts. This solution has the advantage to have minimum interrupt latencies for the real-time events handled this way and the drawback of needing an extra hardware device. As the other solutions presented later on in this paper, an extra communication mechanism is required between the NT and the Real-Time part. The difference here is that a great care

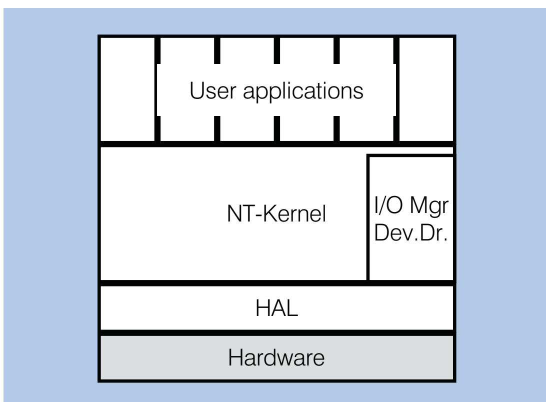

Figure 9. Simplified NT structure

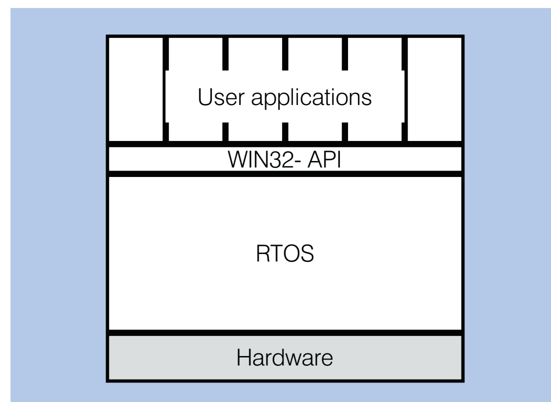

Figure 10. RTOS + Win32 API

should be taken due to the NMI use.

In the next solutions we will talk about the vendors that modify or hook some of the layers: HAL or/and NT-Kernel. How can you modify these?

Microsoft policy is not to accept any modification of the NT kernel. The only modifiable part of the Kernel is the device driver. This is the only possible way to communicate with the Kernel. The Microsoft policy with the HAL is different. The HAL or Hardware Abstraction Layer is the underlying layer of software that interfaces NT to the hardware. The HAL is delivered in source code with a special agreement. As a consequence, some people implement special RT solutions via a Modified HAL or any other combination.

implement a Win32 API on top of another RTOS

Rather than trying to modify the HAL or use another trick to add real-time services to NT some people add the Win 32 API to an OS built expressly for real-time.

The advantages of this approach are:

- there is portability between real-time and non real-time approaches
- small footprint
- real-time behaviour from these real-time OS is known

The drawbacks are:

- Standard NT applications cannot be used
- no mixing possible with NT device drivers - so the whole communication world from NT is not available
- other NT graphic device drivers and application software cannot be used or is difficult to use
- You are limited in your options for future expandability
- You may be forced to use special tools for development and compilations

Commercial Implementation:

At the time of printing of this article the following commercial implementation were known:

- QNX using the Willows library
- VxWorks using the Willows library

make coexist on one processor NT and a RTOS (or part of it)

As processor power increases considerably today, one processor may be enough to run a full application with a real-time part and a non real-time portion. Another solution consists then in combining some real-time OS (or scheduler) together with NT. Two ways of looking to that solution are possible:

- modify the HAL by intercepting interrupts and including a small scheduler or RTOS.
- run NT as one of the tasks on top of a (supervisor) RTOS.

In all cases the HAL has to be modified or at least intercepted. These HAL modifications, such as manipulating the clock or rewriting the way the interrupts are processed, represents an unprecedented, unproved use of the HAL. It creates a non-standard environment and may pose maintainability problems if Microsoft modifies the HAL in a future release. Therefore the difference in the proposed solutions by the different vendors is in trying to make believe that the HAL is modified as less as possible.

Interception of the HAL is also possible via Intel processor tricks. These implementations are therefore only possible on Intel based machines. Interception mechanisms via exception handling on device level are taken away precious computational power. Funda-mentally, all the solutions are quite similar. To help you understand take the following possible implementation (modifying the HAL):

The blue screen code of NT can be seen as simplified supervisor code. By modifying the blue screen code (in the HAL) you can make a simple multitasking mechanism out of it, run NT as one task with lowest priority and run anything else as another task of this "simplified blue screen scheduler". This anything else may be a set of real-time tasks or a full RTOS.

In both cases you need a communication mechanism between the RT and NT part. This can be done at two levels. One is to invent on another Inter Process Communication (IPC) mechanism. An IPC via shared memory is probably the simplest to implement. The drawback of this IPC protocol is the priority level on which the NT User Applications are running. Therefore you can also imagine to interface via the device driver called here DD_Com. This gives you the possibility to give the impression to NT that that the RT sub-system is a device (driver).

The RT tasks are using their own interface to the system, which in most cases will not follow the WIN32 API. The development environment may be the typical one from the used RTOS and may or may not interface to a NT environment for easy development and debugging. The RT tasks will run in a "simple and

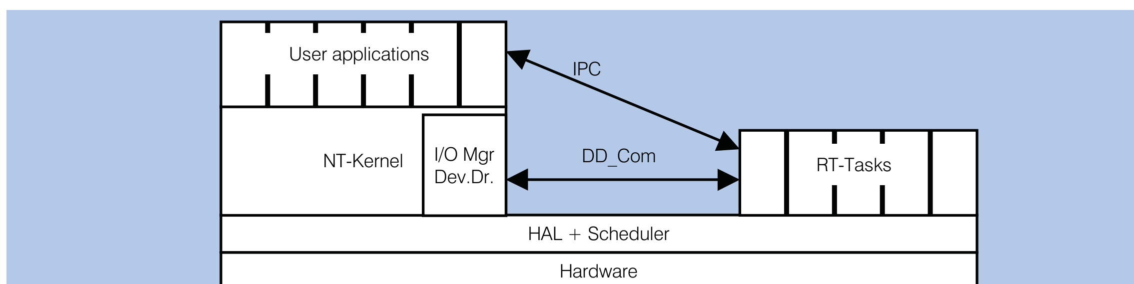

Figure 11. HAL + Scheduler

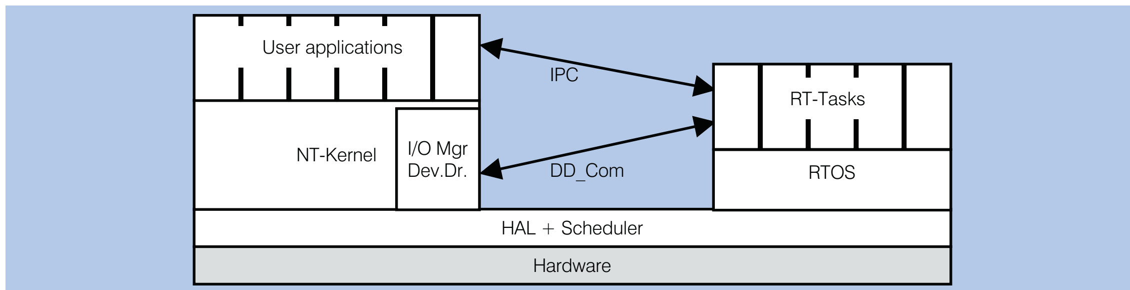

Figure 12. HAL + Scheduler + RTOS

stupid" environment not taking profit of the memory protection mechanism of NT. Special care should be taken to continue running the RT parts even if NT crashed and generates the famous blue screen. The memory footprint is the combination of a minimum NT footprint (8 MB in standard configurations) PLUS the minimum requirement for the RTOS part.

The simplicity of the RT part may lead towards good performance on the RT side, which will depend on the RT Kernel used.

The advantages are:

- Keep the whole NT environment (almost) intact meaning that all software, devices + device drivers for NT can be used (to run the non real-time part of your application). Compatibility with NT is maintained.
- protection for the RT tasks may be included and depends on the protection mechanism of the used RTOS

The drawbacks are:

- non-portability between real-time and the NT environment except if a RT-API is developed inspired from WIN32
- device drivers for NT can not be used in RT part (specific RT driver has to be foreseen for the specific RTOS if predictable responses for this device are required)
- development environment may be complex if a separated environment is needed for the RT tasks
- a lot of task levels and so context definitions may exist. The switching through all these contexts takes time.

Commercial implementations:

At the time of printing of this article the following commercial implementations are known: (only implementation NOT modifying the hardware is considered here)

- IMAGINATION with HyperKERNEL
- RADISYS with INtime with RT API
- VENTURCOM with RTX, KPX and RTAPI

use a multiprocessor architecture

In a multiprocessor architecture, a simple solution consists in having NT on a processor engine (= one or more processors) and the real-time part on one or more other processors. Parallel bus architectures with VMEbus, PCI and PMC , or serial bus architectures with ethernet are possible here. The subsystems fundamentally communicate via an IPC mechanisms and Remote Procedure Calls if necessary.

If an ethernet connection is used, it only works if you can live with a non-deterministic low rate communication between the NT application and the RTOS. In this case, you need a device driver in NT to communicate with your RTOS machine. Parallel architectures may deliver predictable response, but on one condition: that you know in very much detail how your bus behaves in a multiprocessor multimaster approach.

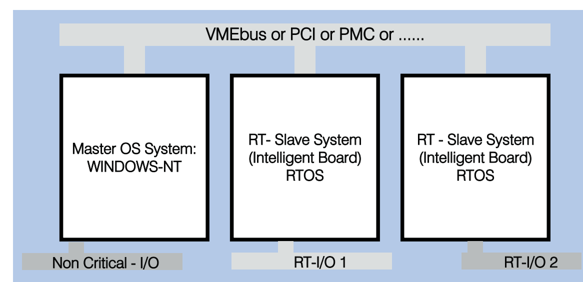

Figure 13. Multiprocessor approach

This is object orientation with not only data and processes encapsulated but also the processor (or hardware).

Advantages

- No modification to each OS.
- Well suited for large complex systems.
- For each subsystem a different RTOS can be chosen adapted to the job.
- Existing subsystem developments (= intelligent I/O boards) can be re-used. They only have to be rebuilt with a VME or PCI (PMC or CPCI) interface if they don't have this interface already.

Drawbacks

- Expensive because 2 or more processors are needed.
- Real-time behaviour depends on the behaviour of the interprocessor communication channel (VMEbus, PCI, PMC etc..) and this should not be neglected! Interrupt behaviour on the bus in such structures is only predictable if well designed and organised.
- Keep development environments running for two (or more) worlds.

Commercial Implementations:

All available intelligent I/O boards running will fall in this category. If you are satisfied with the functionali-

Green Hills Software Ad

ty the board manufacturer is offering you than you have nothing to do. If you want to build such a board yourself or you are not satisfied with the software functionality offered, then you need to develop this subsystem yourself around one or another RTOS. If the board is simple - no RTOS is required - one program and some ISR may do.

CONCLUSION:

There are no miracles in this high tech world. If you want to keep the NT compatibility high, then you will have to pay the overhead price. If you are only interested in some WIN32 API interface, you may live with existing RTOS having this interface.

There is a tremendous demand for these NT & RT combinations. Even if this is not the best solution, the market should follow the customer desires. So the vendors of the different implementations are trying to explain that their solutions are "very fast". In most cases they talk about interrupt latencies, when only one event is involved and the system is slightly loaded. You need to consider the overall response of the system and look to real-time event to task run latencies when the system is overloaded.

As this story is certainly not finished, the Real-Time Magazine evaluation crew will closely follow the evolution. Stay tuned. ■

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. M. TIMMERMAN & J.C. MONFRET, Windows NT as Real-Time OS? Real-Time Magazine, Issue 97/2
2. J. Zalewski, What Every Engineer Needs To Know About Rate-Monotonic Scheduling: A Tutorial, Real-Time Magazine, Issue 95/1
3. P. KORTMANN and G. MENDAL, PERTS Prototyping Environment for Real-Time Systems, Real-Time Magazine, Issue 96/1
4. J.F. READY: Designing for worst case, Real-Time Magazine, Issue 97/3
5. Jeffrey Richter, Advanced Windows, The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95, Microsoft Press, 1995.
6. Microsoft® Windows NT™ Version 3.51 Device Driver Kit Documentation.

REAL TIME
EXPERTISE

Real-Time Consult can assist you with:

- Audits
- Debugging interventions on site
- Consultancy:
  - Architectural and System Design
  - Market & Product Studies and Reviews
- Real-Time Expert Seminars on:
  - Bus Architectures ("The right bus on the right place", VME, PCI, mezzanines, etc.)
  - RTOS (Unix and non-Unix like RTOS, Windows NT, etc.)
  - Software Engineering (analysis, design, imple mentation, testing and debugging of real-time systems)

For more information visit us on the Internet at http://www.realtime-consult.com

Need help urgently? Send an e-mail to info@realtime-consult.com or contact us at 32-2-520.55.77