Thread-based analysis of embedded applications with real-time and non real-time processing on a single-processor platform

Dietmar Prisching
AVL List GmbH
Graz AUSTRIA
dietmar.prisching@avl.com

Bernhard Rinner
Institut für Technische Informatik
Technische Universität Graz
rinner@iti.tu-graz.ac.at

Abstract

The functionality and complexity of embedded systems is steadily increasing. In order to keep time-to-market as short as possible developers apply more and more open architectures, commercial off the shelf (COTS) components and standard platforms. Scalability, real-time (RT) performance and interoperability between RT and non-RT processing are now important requirements of many embedded systems. In this paper, we describe a possible combination of RT and non-RT processing, present a tool for analyzing RT applications and demonstrate the tool's feasibility in the development of a complex embedded system in the automotive domain.

Keywords:
Embedded system; automation system; automotive; context switch; real time; timing analysis

1. Introduction

Both real-time systems (RTS) and automation systems (AS) are classified as computing systems that must react within guaranteed time windows to events in the environment. Predictability and dependability are important requirements for a RTS. Most real-world AS are comprised of a real-time (RT) part and a non real-time (NRT) part. An often used approach to realize a system that requires a RT and NRT combination is to develop the RT and NRT part on separated platforms, e.g., by using dedicated processors for the NRT and RT parts. However, the development of a RTS or AS on multiple platforms is an expensive and tedious task. Since the processor's computing performance has increased significantly over the last years, a single processor platform may now offer sufficient performance to run systems with RT and NRT parts. A single platform solution eases the development and maintenance of the AS and reduces time-to-market and the overall cost of the product.

In this paper we present the *ProfileAnalyzer* - a tool that measures and analyzes thread-based applications with RT and NRT processing on a single-processor platform. The goals of this tool are (i) to support the analysis, modeling and development of RTS, (ii) to monitor an implemented RTS to check whether the interoperability between RT and NRT part is satisfied. These goals are achieved by

- the exact measurement of thread context switches and thread execution times,
- the determination of important performance parameters of the RT and NRT part, and
- an intuitive graphical user interface as well as tools for post processing.

Although the ProfileAnalyzer has been implemented for the combination of the (non real-time) operating system Microsoft® Windows NT/2000/XP and the (real-time) operating system INtime, it is in principle applicable to various combinations of a RT and NRT operating system on a single platform.

We have evaluated our ProfileAnalyzer on the PUMA Open automation system. PUMA Open is targeted for the design and test of engines, transmissions and power trains. It is a large and complex AS running under Windows NT/2000/XP and INtime. Its basic functionality includes safety monitoring, data acquisition and storage as well as control and automation functions.

The remainder of this paper is organized as follows. Section 2 presents related work and the state of the in combining RT and NRT processing. Section 3 introduces the Windows real-time extension INtime. Section 4 describes the functionality and implementation of the ProfileAnalyzer. Important measurements and results from the ProfileAnalyzer are presented in Section 5. Finally, Section 6 concludes this paper with a summary and a discussion on further work.

2. Combining RT and NRT Processing

There is a large demand for real-time (RT) and non real-time (NRT) combinations in the market [DED]. In a typical automation system (AS) the RT part is responsible for controlling, data acquisition, signal conditioning and monitoring. The NRT part is responsible for data post processing, visualization, data persistence, system parameter settings and so on. In general, the NRT part is mostly used for the Graphical User Interface (GUI). Consequently, most NRT systems are based on up-to-date general-purpose operating system such as Microsoft Windows. In this context an important parameter is the *interoperability* which is defined as the ability to run NRT applications along with RT applications.

The life cycle of an AS is typically several years. In order to incorporate new peripherals such as measurement devices, the AS must be designed as an open platform. This means that the AS must support various interfaces (variety of standard interfaces). Application programming interfaces (API) are also necessary, which facilitate the connection to specific software-applications and tools to the AS. They also ease the integration of third-party applications.

Resulting from the requirements for RTS and AS respectively, the requirement for a RT-NRT combination can now be summarized as follows:

1. The RT part must react within precise time constraints to events in the environment. A real-time system must responds in a (timely) predictable way to unpredictable external stimuli arrivals.
2. It has to be guaranteed that the NRT part does not influence the RT part. In other words the execution of the RT part is preferred over the NRT part.
3. Due to the requirements of the NRT part, up-to-date NRT parts should support third-party applications in a wide range.
4. The RT and the NRT parts should be executed on a single platform.

The Microsoft® Windows OS includes Windows NT (4.0) and the newer versions of the NT technology Windows 2000 (5.0) and Windows XP (5.1). There exist several approaches to establish real-time systems (RTS) based on Windows NT. Some approaches are based on a relaxation of the real-time requirements and are, therefore, not applicable to AS [BAR98], [DED]. Starting from these limitations, the industry is offering today solutions to give NT a real-time capability. There are extensions to the general-purpose operation system Windows NT/2000/XP available that are able to guarantee hard real-time constraints. One of these approaches is to combine the NRT operating system Windows NT and a RT operating system (or parts of it) on a single processor.

The advantages of this approach are:

- Almost the entire Windows NT environments is kept unchanged meaning that all software, including devices, device drivers for Windows NT can be used (to run the NRT part of the application). The compatibility with Windows NT is maintained.
- Protection for the RT tasks may be included and depends on the protection mechanism of the used RTOS.

The drawbacks are:

- To establish data exchange between the RT and the NRT part, a dedicated RT-API (inspired from the WIN32 API) has to be developed.
- Device drivers for NT can not be used in the RT part. Thus, specific RT drivers have to be included for the specific RTOS if predictable responses for this device are required.
- The development process may be complex if a separated environment is needed for the RT tasks.
- A lot of task levels and context definitions may exist. The switching through all these contexts takes time.

The following commercial implementations are currently known:

- *INtime* from TenaSys is a standalone RT operating system and described in Chapter 3.
- *RTX* from VenturCom is based on a different approach. While INtime is a standalone OS completely separated from NT, RTX is highly integrated into the NT executive. The RTX subsystem and all its applications are implemented as Windows NT device drivers. RTX adds a real time subsystem (RTSS) to Windows NT. RTSS is implemented as an NT device driver.

RTSS is conceptually similar to other Windows NT subsystems such as Win32 and POSIX, WOW and DOS in that it supports its own execution environment and API [RTX].

- The *Hyperkernel* approach from Imagination Systems is a compromise between the RTX and INtime approach. As opposed to RTX, the Hyperkernel subsystem does have its own memory space, which is protected from Windows NT. However, the hardware tasking mechanism is not used like it is in INtime. The Hyperkernel subsystem runs in the same context as the NT executive and uses the same system stack. Like RTX the advantages of such a structure are very fast switches between RTX and Windows NT [HYP].

[OBE99], [OF99] and [OBE00] show how NRT performance of a dual Windows NT – INtime system is affected by the addition of a real-time load. [OBE99] measures the affect of increasing the real-time load on Windows NT processing. The Windows NT performance was measured by the WinBench Suite. [OBE99] evaluated two RT-processing scenarios:

- Constant: A single thread uses the Whetstone synthetic benchmark to create a periodic constant RT load. The thread runs every 100 milliseconds for durations of (10, 20, 30…) ms. NT continues to operate even when it is suspended for a duration as long as 90 milliseconds, i.e., the RT load is 90 %. However, at this point the operation of the Graphical User interface (GUI) is severely affected. The results show also that the RT/NRT interoperability performance is very close to 100% for all cases.
- Distributed: Multiple threads are used to create an evenly distributed RT load. The duration of any thread does never exceed 1ms. Most WinBench performance metrics are lower for a distributed RT load than for a constant RT load, due to the fact that there are more context switches in the distributed case. The CPU WinBench performance is affected most; its worst overhead is 13.5%. There is also a difference in the GUI behavior for the two different types of RT load. The processing requirements needed to maintain GUI performance, i.e., smooth mouse operation and flicker free updates of the screen, is fairly minimal. This is supported by the fact that the subjective GUI performance is relatively unaffected by a distributed load as great as 90%.

3. The Real-Time Extension INtime

INtime uses the hardware task management facilities of the Intel processor architecture when the processor runs in the protected mode. Both NT and INtime are standalone operating systems. Each OS runs within a single hardware task. The Operating System Encapsulation Mechanism (OSEM) is then responsible for the scheduling of the NT and INtime hardware tasks. Therefore, INtime modifies the Hardware Abstraction Layer (HAL) of NT. The advantage of this approach is security and stability. Since INtime runs in a single hardware task, it runs in its own environment completely separated from NT. This protects critical real-time applications from NT crashes. Moreover, INtime application processes execute in user mode (ring 3) which makes that buggy applications are unlikely to render INtime unstable.

The drawback of this approach is performance degradation – switching between NT and INtime can take more than 300 CPU cycles. Whenever NT needs to be preempted because an INtime thread is ready to run, this delay needs to be taken into account. RTX 4.2 and Hyperkernel 4.3 are faster in switching between NT and them, but they do not offer the same level of protection as INtime does.

INtime is a multi-process, multi-thread kernel. Priority based preemption is used for scheduling threads. Round robin scheduling policy is used for equal-priority threads. 255 priority levels are available from which the upper 128 are used by the system for interrupt handling and other critical threads. The lowest-priority thread within the INtime system represents NT and the applications running under NT. When no other thread is ready to execute, the lowest-priority thread is scheduled by INtime.

There are three ways to service an interrupt by INtime.

- Using a handler alone: The interrupt handler is a function executing in the context of the interrupted thread. During the time the interrupt handler is executed, all interrupts are disabled and no thread with a higher priority can preempt the handler. An interrupt handler alone can process only interrupts that require very little processing time.
- Using a handler/thread combination: An interrupt handler/thread combination provides more flexibility. The handler signals to a corresponding interrupt thread. The RT kernel assigns an

interrupt thread's priority, which is based on the interrupt level associated with the handler. Higher-level interrupts are enabled when executing the interrupt thread.

- Thread rescheduling: The third method to service an interrupt is to provoke thread rescheduling. Within the handler scheduling must be disabled. The handler can provoke thread rescheduling by signaling a semaphore or posting to a mailbox. Then the thread that waits for the semaphore or mailbox will be scheduled. By this method the flexibility like using handler/thread combination is possible but the priority for the interrupt thread can also be chosen explicitly [INT1.2], [INT98].

4. ProfileAnalyzer

4.1 Functionality

The ProfileAnalyzer profiles thread context switches and dispatching of threads, respectively. This is realized as follows. An *instrumentation function* writes the dispatch time (in CPU ticks) together with the unique OS thread identifier into a shared memory location (see Figure 4-1 (1)). Since the shared memory is organized as a ring buffer, the data must be saved (copied) at appropriate time intervals (see Figure 4-1 (2)). The length of the time interval depends on the size of the ring buffer and the RT-processing load. The instrumentation function is attached to the INtime scheduler and has been implemented as a ring 0 (kernel-mode) program.

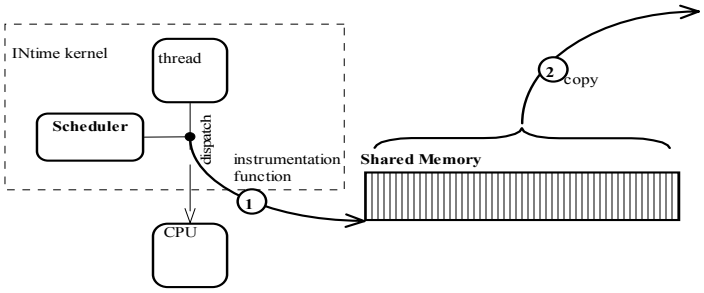


Figure 4-1: The instrumentation function of the ProfileAnalyzer.

As mentioned above, the lowest-priority thread within the INtime system represents Windows and the applications running under Windows. This Windows thread is represented by the name *NT_TASKREAD* within the INtime system. When no other INtime thread is ready to execute, the *NT_TASKREAD* is scheduled by INtime.
Figure 4-2 shows a sample task profile depicted as a Gantt chart. As a result of the measurements stored in the ring buffer, the timing parameters of the INtime tasks (τ_i) can be determined. Examples for these parameters are the average computation time (C_i) of an INtime task and its time period (T_i).

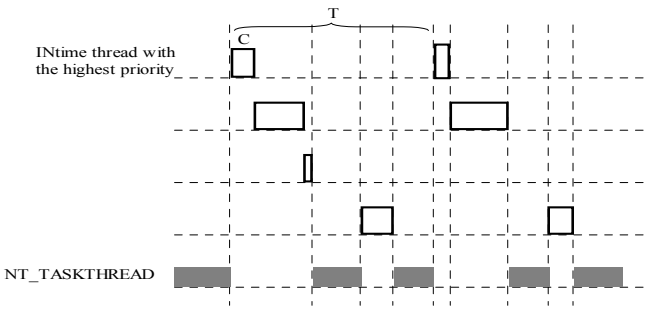


Figure 4-2: Schematic Gantt chart View from Windows/INtime thread interoperability

The timing parameters are determined in the following way: The ProfileAnalyzer observes the system for a certain time interval ΔT, which can be set by the user. Each invocation of a task within ΔT is logged and so the minimum, maximum and average computation time (C_i) of the task can be calculated. The number of invocations corresponds to the time period of the task. However, it is

possible that an invocation of a task is preempted by tasks with higher priority. In this case, the user can set the time period of a task and the execution time of the task without any interference is calculated.
An important performance metric calculated from the ProfileAnalyzer is the *CPU usage (utilization)* that results from a certain RT load. Given a set Γ of n periodic tasks, the processor utilization factor U is the fraction of processor time spent in the execution of the task set. Since C_i/T_i is the fraction of processor time spent in executing task τ_i, the utilization factor for n tasks is given by [BUT97]:

Equation 4-1: $$U = \sum \frac{C_i}{T_i}$$

A primary goal of the ProfileAnalyzer is to visualize the measured data in a usable manner. The thread results are displayed in a practical Gantt chart view diagram (see Figure 4-3 (1)) and it is possible to zoom in and out at the time axis of the diagram (see Figure 4-3 (2)). Furthermore, thread properties like execution times of its invocations, minimum, maximum and average computation time (C_i), idle time as well as cyclic time (T_i) can be viewed (see Figure 4-3 (3)). The calculated performance metrics of the RT-processing like the processor utilization factor U can be also displayed (see Figure 4-3 (4)). In addition it is possible to compare results from different measurements. Measured data can be stored and reload back into the ProfileAnalyzer.

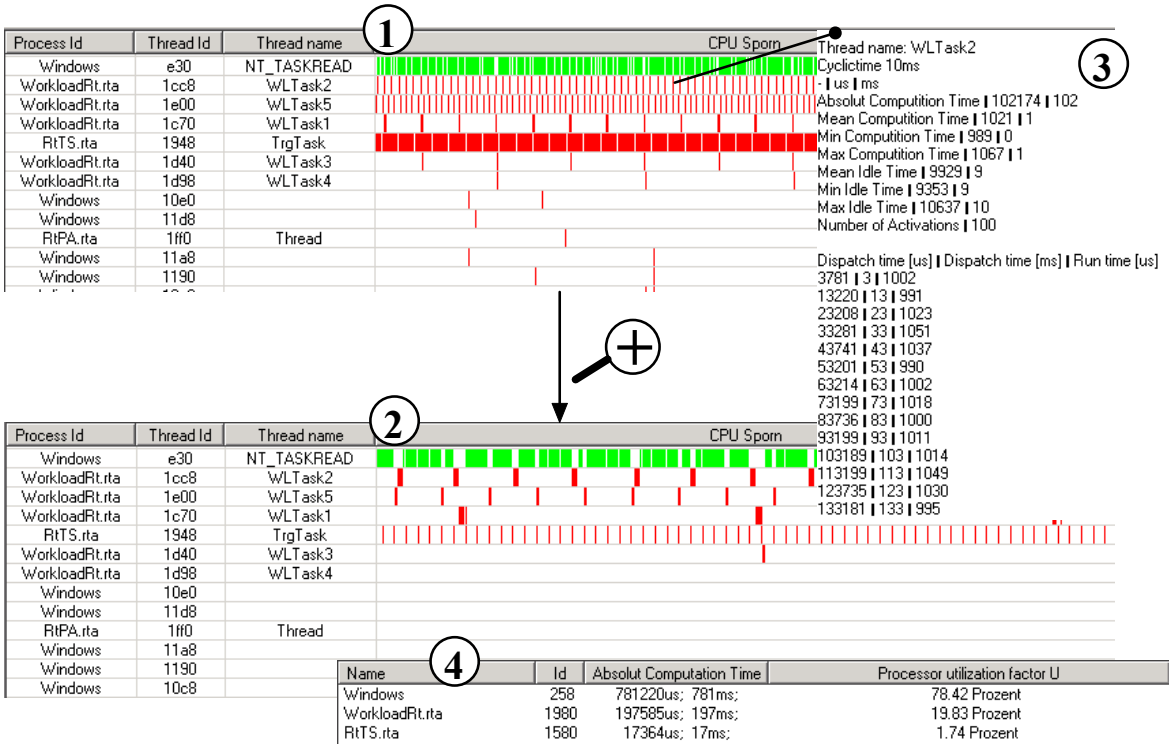


Figure 4-3: ProfileAnalyzer Snapshots

4.2 Implementation

The ProfileAnalyzer has been completely specified using the unified modeling language (UML). A use case and a software design description (SDD) model have been derived from that specification. Figure 4-4 depicts the component view. The RtProof.rta is an INtime Ring 0 program implemented by [TenaSys]. It includes the instrumentation function described above. The RtPA.rta is an INtime program that copies measured thread data in a deterministic manner. It further provides important INtime objects information about the analyzed RT-processing. It is implemented in C using the Microsoft® Visual Studio 6.0. The ProfileAnalyzer.exe is a Windows program implemented in C++

based on the Microsoft Foundation Class (MFC). It is responsible for the visualization, the post processing and the storage of the measured data. The communication between the RtPA.rta and the ProfileAnalyzer.exe is performed by the INtime mailbox mechanism.

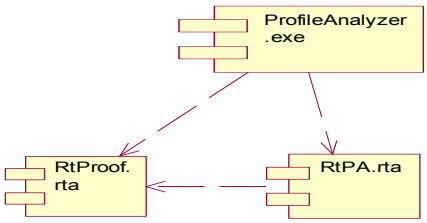


Figure 4-4: Component view of ProfileAnalyzer.

5. Measurements and Results

5.1 Deviation of results from ProfileAnalyzer and Windows System-Monitor

The system-monitor (Performance Monitor) from Microsoft® Windows is also a tool that allows the monitoring and displaying of various performance parameters. The CPU usage resulting from an INtime load can also be determined by the Windows system-monitor. This is realized by providing a Windows performance counter object (RT Kernel CPU usage) that is accessible from the system-monitor. However, it was found that the measured CPU usage values from the system-monitor deviates from ProfileAnalyzer values. This is due to the fact that the ProfileAnalyzer charges the context switch time between the operating systems to Windows (*NT_TASKREAD,* see
Figure 5-1). On the contrary, the INtime performance counter value contains the context switch time (see
Figure 5-1). Therefore, the deviation between the system-monitor and the ProfileAnalyzer results increases with higher OS context switch rate.

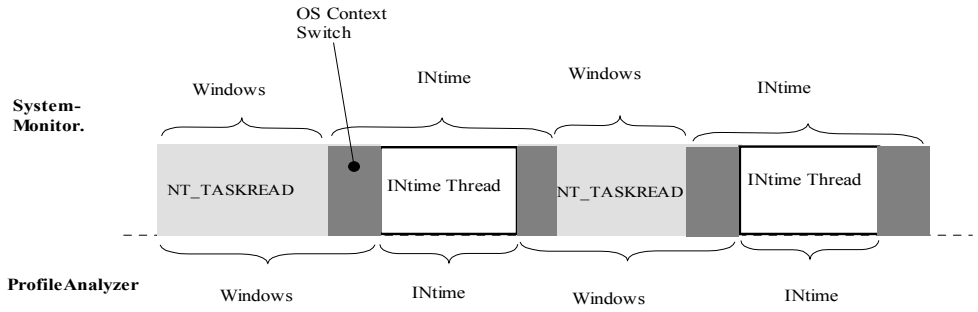


Figure 5-1: Different methods for determining the CPU usage by the ProfileAnalyzer and the Windows system-monitor.

To determine the deviation of the CPU usage values between the ProfileAnalyzer and the system-monitor we carried out the following measurement. We implemented a simple INtime program (ParallelInterrupt.rta) that only handles the parallel interrupt (interrupt request) IRQ 7 with an interrupt handler. Thus, at each IRQ 7 the interrupt thread from ParallelInterrupt.rta becomes ready. We triggered the parallel interrupt IRQ 7 by a multi frequency generator (MFG). Thus, by changing the MFG frequency the OS context switch rate was modified. The measured deviation (CPU usage (%)) between the ProfileAnalyzer and the system-monitor in dependency to the OS context switch rate is depicted in Figure 5-2. The measurement was performed on a Pentium II with 434 MHz.

5.2 Determination of the OS Context Switch Time

The deviation between the ProfileAnalyzer and the system-monitor results is derived from the OS context switch time (see above). Therefore, the values from Figure 5-2 deliver the time for an OS context switch (CS_{WI}) between Windows and INtime. In our configuration we determined CS_{WI} with 5µs. According to [TenaSys] a switch from Windows to INtime needs about 300 CPU cycles. The context switch from INtime to Windows can takes longer if the floating point unit (FPU) registers must be restored. The FPU registers are saved at the first access from an INtime thread.

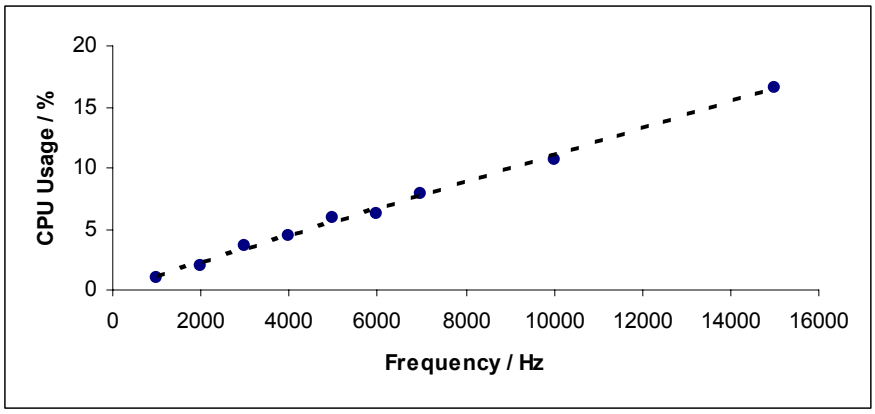
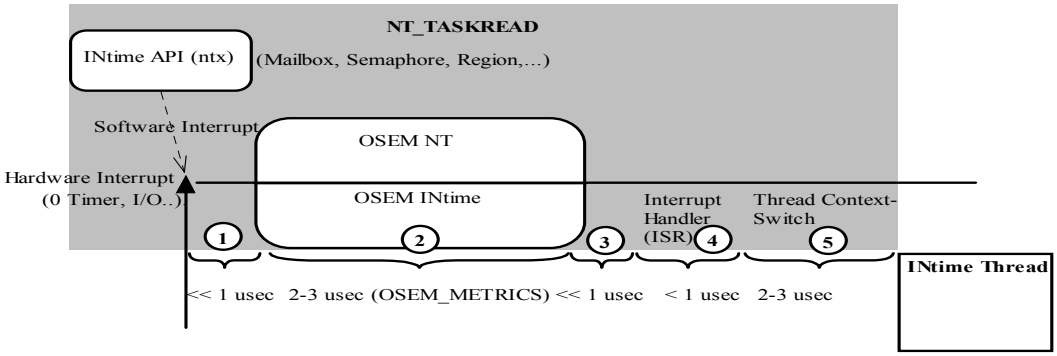


Figure 5-2: CPU usage deviation between ProfileAnalyzer and system-monitor

The duration of CS_{WI} as 5µs can be explained as follows. First, Windows is active and an INtime thread becomes ready, e.g. by a timer, I/O interrupt, a Windows INtime API call or a software or hardware interrupt. Therefore, the Operating System Encapsulation Mechanism (OSEM) entry point is called. The time between the interrupt in Windows and OSEM (Windows environment) switch path entry is very small (<< 1µs) (see
Figure 5-3 (1)). Next the switch by the OSEM (switch of the OS hardware tasks) occurs. INtime provides the performance metrics (OSEM_METRICS) for the OSEM switch[1]. In our configuration, the average value for the OSEM switch was 2.376 µs (see
Figure 5-3 (2)). Next a switch from the OSEM (INtime environment) to the IRQ 7 interrupt service routine (ISR) occurs which requires also less than a 1µs (see
Figure 5-3 (3)). The time for the ISR itself depends on the interrupt handler. According the developer guide the interrupt handler code should be small (see
Figure 5-3 (4)). This completes a full thread context switch from the *NT_TASKREAD* to the ready INtime. The full context switch requires about the same time as going through the OSEM. In our case it is between 2 and 3µs (see
Figure 5-3 (5)).



[1] This feature is currently available only in an INtime OS prototype. It should become available in INtime version 2.20.

Figure 5-3: Overview Windows – INtime OS context switch

5.3 Windows Performance affected by RT load

RT-processing does affect the performance of Windows. For example, if Windows doesn't get the control within 7ms its Graphical User interface (GUI) is severely degraded [TenaSys]. Generally, the Windows – INtime interoperability performance should be very close to 100%. This is true for a RT load that generates a low OS context switch rate (see [OBE99] for constant RT-processing). However with rising OS context switch rates the Windows – INtime interoperability performance isn't 100%. [OBE99] determined for the CPU WinBench a worst overhead of about 13.5%. For clarification we also performed some measurements with the Windows benchmark Business Graphics WinMark 99. For this experiment we used our ParallelInterrupt.rta configuration (see above) to generate a RT-processing at different frequencies. Furthermore, we generated different RT loads by increasing the execution time of the interrupt thread. From Table 5-1 (RT-processing frequency is 2 kHz) it can clearly be seen that the overhead is independent of the load itself. The value for the average overhead for a 2 kHz RT load was determined with 5.75 %. Figure 5-4 shows the trend for the overhead dependent on different OS context switch rates. It clearly indicates that with rising OS context switch rate the overhead for Windows - INtime interoperability increases.

Table 5-1: Windows - INtime interoperability performance overhead of RT-processing at 2 kHz.

INtime CPU Usage / % (system-monitor)	Business Graphics WinMark 99	Overhead / % *
0	100	0
10	82,07253886	7,92746114
16	77,72020725	6,279792746
22	70,05181347	7,948186528
30	67,25388601	2,74611399
34.2	58,03108808	7,768911917
52.2	40,7253886	7,074611399
75	19,68911917	5,310880829

* Average Overhead (2 kHz) = 5.75 %

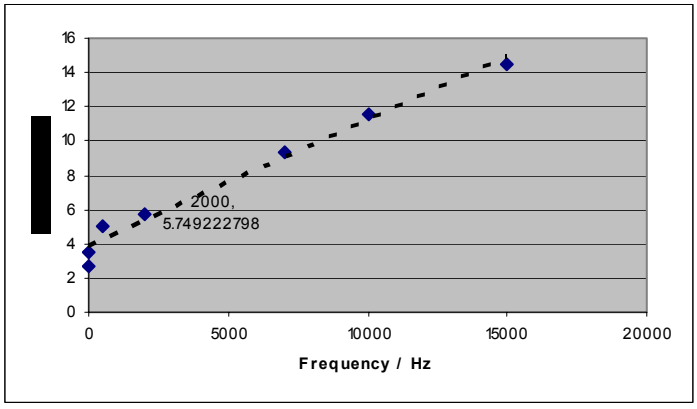


Figure 5-4: Windows - INtime interoperability performance overhead in dependency to OS context switch rate.

5.4 Example: PUMA Open Rate Monotonic Scheduling

An important part of the RT-processing from the AS PUMA Open is scheduled according to the rate monotonic (RM) policy. The RM scheduling algorithm is a simple rule that assigns priorities to tasks according to their periods.
Figure **5-5** presents a ProfileAnalyzer snapshot of the PUMA Open RT-processing and it obviously depicts the RM scheduling policy.

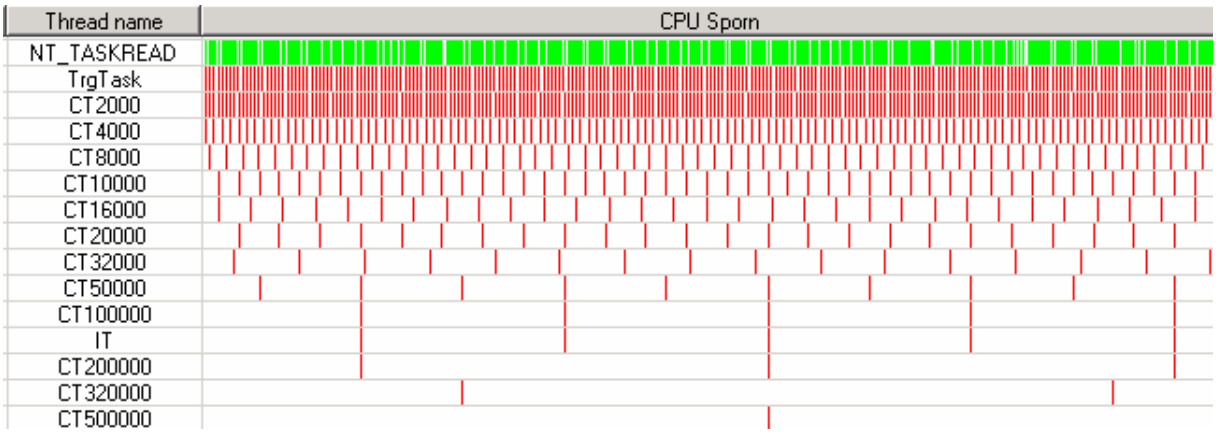


Figure 5-5: Rate Monotonic thread scheduling analyzed by the ProfileAnalyzer

5.4 Example: Basic PUMA Open RT-Processing

Figure *5-6* shows a ProfileAnalyzer snapshot from a basic PUMA Open RT-processing. The large number of tasks demonstrates the importance of a tool that allows the thread-based analysis. Such a tool is especially important during the development phase where tasks may change frequently added and deleted and their influence on the overall task set must be well understood.

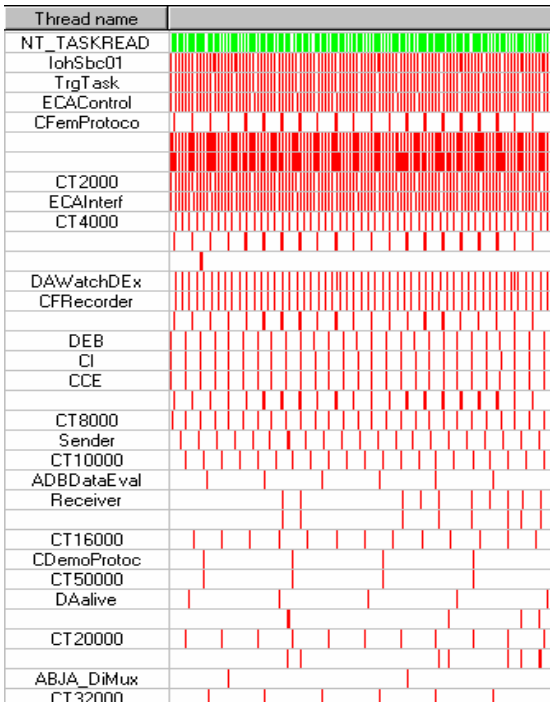


Figure 5-6: Extract from a basic PUMA Open RT-processing analyzed with the ProfileAnalyzer

6. Discussion

We have presented the ProfileAnalyzer – a tool that measures and analyzes (thread-based) applications with RT and NRT processing on a single-processor platform. Although the ProfileAnalyzer has been implemented for the NRT operating system Microsoft® Windows NT/2000/XP™ and its RT extension INtime, results from this research are applicable for a general RT and NRT combinations. These results include:

- The context switch time between the RT and the NRT operating system is a significant performance factor. Particularly at high OS context switch rate it must be considered as a non-negligible overhead (see Chapter 5.1).
- NRT services, e.g., the graphical user interface (GUI), and NRT performance respectively are influenced by the RT-processing. If the NRT part is suspended too long due to RT-processing, the NRT services are severely affected (see Chapter 5.3 and [OBE99]).
- The OS context switch rate affects the RT and NRT interoperability performance. At higher OS context switch rate the RT and NRT interoperability performance isn't 100% (see Chapter 5.3 and [OBE99]).

With INtime 2.20 a system analyzer will be released (InScope). InScope provides an analysis of threads, interrupts and INtime API calls. Comparison from the ProfileAnalyzer and InScope shows that the displaying of analyzed threads is more comfortable with the ProfileAnalyzer. Particularly at measurements over a longer time interval the ProfileAnalyzer provides a better overview. However, the InScope isn't only focused to analyze threads. Experience shows that the InScope and the ProfileAnalyzer are two tools that mutual complemented it.
Several functional extensions are considered for the ProfileAnalyzer. Especially a software application interface is specified. So, it is possible to start and stop the ProfileAnalyzer from another application to record certain time ranges.

References

[BAR98] BARIL, A., 1998: Using Windows NT in Real-time Systems, Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium

[BOL93] BOLCH, G., VOLLATH, M., 1993: Prozessautomatisierung, Stuttgart

[BUT97] BUTTAZZO, G. C., 1997: HARD REAL_TIME COMPUTING SYSTEMS, Predictable Scheduling Algorithms and Applications, London, Kluwer Academic Publishers

[DED] http://www.dedicated-systems.com

[HYP] Dedicated Systems Experts NV. RTOS Evaluation Project "Hyperkernel 4.3", http://www.dedicated-systems.com

[INT1.2] Real Time Magazine Real-Time Consult "INTIME 1.20"; http://www.dedicated-systems.com

[INT98] INtime Software Overview Guide, RadiSys

[OBE99] OBENLAND, K., ROSEN, L., 1999: The Interoperability Between Real-time and Non-real-time Processing on a Windows NT Platform

[OBE00] OBENLAND, K. M., ROSEN, L. H., 2000: The Performance Trade-offs of Implementing a Large Scale Real-time Application Using the Windows NT Operating System, Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium, 2000

[OF99] OBENLAND, K. M., FRAZIER, T., KIM, J. S., KOWALIK, J., 1999: Comparing the Real-time Performance of Windows NT to an NT Real-time Extension

[RTX] VenturCom User's Guide 4.1 (5.0)

[TENASYS] www.tenasys.com