Doc: **EVA-2.9-TST-CE7-x86-01**	Issue: **4.1 on 6-Jun-2012**	Tests Date: **May - June ,2011**

Behavior and performance evaluation of Windows Embedded Compact 7 on x86

Authors

Luc Perneel (1, 2), Hasan Fayyad-Kazan(2) and Martin Timmerman (1, 2, 3)
1: Dedicated Systems Experts, 2: VUB-Brussels, 3: RMA-Brussels

http://download.dedicated-systems.com E-mail: info@dedicated-systems.com

EVALUATION REPORT LICENSE

DOCUMENT CHANGE LOG

Issue No.	Revised Issue Date	Para's / Pages Affected	Reason for Change
1	May 28 2011	All	Initial draft
1.01	May 29 2011	All	Comments
2.0	May 30 2011	All	QA approved
3.0	July 28, 2011	All	Including new calibration test (making comparison possible with ARM tests)
3.1	Sept 13, 2011	All	Final Report
3.2	Sept 19, 2011	All	Fonts included
3.3	December 25, 2011	All	Change pages' header
4.0	Feb 18, 2012	All	Add MS comments
4.1	June 6, 2012		Add vendor comments

1 Document Intention

1.1 Purpose and scope

This document presents the quantitative evaluation results of the **Windows Embedded Compact 7** OS on x86-based platform.

The layout of this report follows the one depicted in "The OS evaluation template" [Doc. 4]. The test specifications can be found in "The evaluation test report definition" [Doc. 3]. For more detailed references, See section "Related documents" of this document. These documents have to be seen as an integral part of this report!

Due to the tightly coupling between these documents, the framework version of "The evaluation test report definition" has to match the framework version of this evaluation report (which is 2.9). More information about the documents and tests versions together with their corresponding relation between both can be found in "The evaluation framework", see [Doc. 1] in section "Related documents" of this document.

The generic test code used to perform these tests can be downloaded on our website by using the link in the "related documents" section.

1.2 Test framework used: 2.9

This document shows the test results in the scope of the evaluation framework 2.9. More details about this framework are found in Doc 1 (see section "Related documents").

1.3 Conventions

Throughout this document, we use certain typographical conventions to distinguish technical terms. Our used conventions are the following:

- ***Bold Italic*** for OS Objects
- **Bold** for Libraries, packets, directories, software, OSs...
- `Courier New` for system calls (APIs...)

Related documents

These are documents that are closely related to this document. They can all be downloaded using following link:
http://www.dedicated-systems.com/encyc/buyersguide/rtos/evaluations

Doc. 1 The evaluation framework
This document presents the evaluation framework. It also indicates which documents are available, and how their name giving, numbering and versioning are related. This document is the base document of the evaluation framework.
EVA-2.9-GEN-01 Issue: 1 Date: April 19, 2004

Doc. 2 What is a good RTOS?
This document presents the criteria that Dedicated Systems Experts use to give an operating system the label "Real-Time". The evaluation tests are based upon the criteria defined in this document.
EVA-2.9-GEN-02

Doc. 3 The evaluation test report definition
This document presents the different tests issued in this report together with the flowcharts and the generic pseudo code for each test. Test labels are all defined in this document.
EVA-2.9-GEN-03 Issue: 1 April 19, 2004

Doc. 4 The OS evaluation template
This document presents the layout used for all reports in a certain framework.
EVA-2.9-GEN-04 Issue: 1 April 19, 2004

Doc. 5 Windows Embedded Compact 7, Theoretical evaluation.
This document presents the qualitative discussion of the OS
EVA-2.9-OS-CE-7 Issue: 1 May 20, 2011

2 Introduction

This chapter talks about the OS that we are going to test and evaluate, and the hardware on which the under testing OS will be employed to be tested.

For a more in depth discussion about the positive and negative points, the reader should also read the theoretical evaluation report.

2.1 Overview

Releasing a new OS with a different name (changed from **Windows CE** to **Windows Embedded Compact 7**) does not mean that we are up with a new OS! Such naming change was mainly done for marketing purposes, as there were no fundamental changes in the OS itself!

Further in the document, the full name "**Windows Embedded Compact 7**" or the short names "**Compact 7**" and "**CE7**" will be used.

2.2 Evaluated (RTOS) Product

This section describes the OS that Dedicated Systems tested using their Evaluation Testing Suite, and the hardware on which this OS was running during the testing.

2.2.1 Software

The RTOS that will be evaluated and tested is **Windows Embedded Compact 7**. This OS was launched by Microsoft Corporation at the beginning of 2011. In fact, this OS "**Windows Embedded Compact**" is the successor of **Windows CE6R3**.

The tests for evaluating this OS were done in March 2011 which is the date when this OS was released as a manufacture release.

2.2.2 Hardware

The hardware used for testing this OS version is Pentium MMX 200MHz platform. Indeed, it is an old platform but with such platform, the performance can be compared for over a decade.

As **Compact7** does not run anymore on the Pentium MMX, we had to choose a more recent CPU for our tests. Pentium II running at 233MHz was chosen. Besides that it has a little higher clock frequency, it has also a 512KB L2 cache compared with the previous generation Pentium MMX (which has none).

All the tests were executed on hardware with the following characteristics:

- Motherboard: Intel AL440LX with a 66MHz PCI bus
- BIOS: 4A4LL0X0.86A.0031.P14
- CPU: Intel Pentium II 233 MHz (with 16KB Data and 16KB Instruction L1 Cache). 512KB L2 Cache.
- RAM: 192 MB
- Network interface card: The Realtek RTL8139C(L)
- VMETRO PCI exerciser in PCI slot 3 (PCI interrupt level D, local bus interrupt level 10)
- VMETRO PBT-315 PCI analyser in PCI slot 4.
- External and CPU internal cache was enabled during the tests.

3 Evaluation results summary

Following is a summary of the results of evaluating **Windows Embedded Compact 7**, released by Microsoft Corporation, Inc.

3.1 Positive points

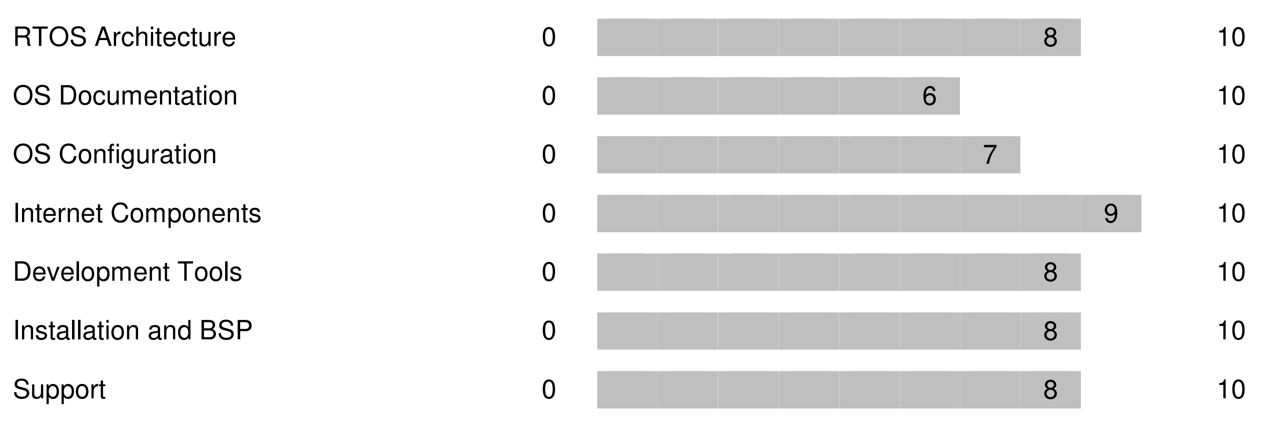
1) All protection primitives use priority inheritance, which is a major plus for achieving real-time behavior
2) Good debugging tools: Available also for kernel/driver debugging.
3) Very easy to install and to set-up a target (from templates).
4) Provides the same flexibility as a 32-bit general purpose OS

3.2 Negative points (see Microsoft's comments in section 3.4)

1) The operating system documentation has taken a step backwards compared with the previous versions. A lot of background information is removed (***see MS comments***).
2) Customizing the kernel and adding custom drivers (BSP) stays a daunting task once you go away from the default configurations.
3) The remote tool has been changed since last version. We noticed two issues, the more important of which is that there is no officially-supported method to include the remote tools within a device image using Platform Builder. Additionally, we noticed during our testing that establishing a connection between the tools and the target took in excess of a minute, which was longer than our expectation (***see MS comments***).

3.3 Ratings

For a description of the ratings, see [Doc. 3].



3.4 Vendor Comments

Following are the comments of Microsoft on the negative points:

- **For point 1** Microsoft notes that documentation is a focus for the next release, and the product team plans to bring forward any relevant content from earlier releases, which will be identified as still applicable to the current release.
- **For point 3** Microsoft notes that the ability to add the remote tools to a device image using Platform Builder is by design, as generally a finished device's final image would not normally include debug support. Additionally, because some devices won't have a .CAB installer, making installation of the remote tools a challenge. They are investigating now how to provide this support in a future release of Platform Builder. Microsoft also notes that the Compact Product Team was unable to reproduce the delayed connection time experienced by Dedicated Systems but will continue to investigate whether connection time is a persistent issue.

4 Test Results

Test Results	0	9	10

If we are looking only to absolute values, most calls are a bit slower than the direct competition. However for real-time behaviour, the worst case delay is more important and this is very well controlled by **Windows Embedded Compact 7.**

4.1 Calibration system test (CAL)

"Calibration tests" are performed to calibrate the tracing overhead compared with the processing power of the platform. Such tests are important to understand the accuracy of the measurements done in scope of this report, and for measuring the processing power of the platform. This calibration permits comparison with the results on other platforms.

4.1.1 Tracing overhead (CAL-P-TRC)

"Tracing overhead test" calibrates the tracing system overhead. It is more related to the hardware than the OS because its aim is to correct the measured time values.

In the rest of the document, the tracing overhead is subtracted from the obtained results.

Tracing accuracy depends here on the PCI clock (33 MHz), as this is the minimum time frame that can be detected. In general, the results in this report are correct to +/- 0.2 µseconds. Therefore the results shown in the tables are rounded to 0.1 microseconds.

4.1.1.1 Test results

Test	result
Average tracing overhead	210 nsec
minimum tracing overhead	210 nsec
maximum tracing overhead	210 nsec

4.1.2 CPU power (CAL-P-CPU)

The “CPU power” test calibrates the CPU performance and the memory bandwidth of the used platform. This test is measured in different situations, starting from the situation where code and data are cached, until the situation where neither code nor data are cached. With such different situation tests, the effects of the cache can be calculated.

We have been seriously reworking this test lately. The CPU test uses only one data address; The non-cached version is about 128KB in size (instructions), while the cached version uses a loop (a bit unrolled to have a small loop overhead but so it fits in the L1 I-cache and it uses only two data words). The instruction cache test is done twice:

- The instructions have not been mapped yet (leading to TLB exceptions and page faults)
- There will not be any page faults (TLB exceptions will still happen).

This gives us some indication about the impact of page faults.

For this specific ARM platform, we used a factor 5 for the test, so that 5 x 128KB = 640KB is larger than the L1/L2 caches. After the test, we divide the results by 5, in order to be capable to compare the results with other platforms

Further, we divided the data cache tests into a read test (reading content of a large array in non-cached case, and read a small array in a loop in the cached case) and a write test. Remark that we flush the caches in between the tests.

This rework shows that a worst-case / best-case scenario can cause significant performance impacts, something that in reality will almost surely never be that large (or you should be able to run everything using only L1 caches).

The impact of either having the code in the I-Cache or not, has serious effect on the results of the tests.

Remark that the results of such tests will depend also, to a high extent, on the cache organization:

- Number of ways
- Line size
- Number of address bits used for index
- Virtual or physical addresses used as index.

4.1.2.1 Test results

The results for **Compact 7** on the Pentium II 233MHz are shown below (averaged over 10 test runs):

Test	no cache	cached	cache effect
CPU test: first load.	604.4 us		
CPU test: ICache effect	539.9 us	215.8 us	2.4
MEM write test	333.6 us	309.2 us	1.1
MEM read test	253.2 us	166.4 us	1.5
Average caching effect (CPU and MEM)			1.7

Here are some conclusions regarding the Pentium II 233MHz:

- Caching of instructions has a huge impact! This is logical because for each instruction, memory has to be fetched containing this instruction. When handling data, you will always have some instructions without data access (register manipulations and operations) which are not impacted by the data cache.
- Initial load has some impact on performance: this can be caused by L2 and by extra TLBs.
- Caching does NOT have a huge impact on data writes: writes can be postponed, so they do not block the next instructions in the pipeline from executing.
- Caching has much more impact on data reads: instructions have to wait until the data becomes available. This will take longer if this data is not cached compared to the case where it is. Remark that even if the data is cached, it might take somewhat longer to get compared to write case (due to a postponed write).

Clearly, interrupt handlers and other code with real-time requirements can be much slower if they are not in the cache.

The results cannot be compared with other tests that we did on the same platform. Even if the same code is used, these figures can be different depending on compiler optimizations and compiler versions.

4.2 Clock tests (CLK)

"Clock tests" measure the time needed by the operating system to handle its clock interrupt. On the tested platform, the clock tick interrupt is set on the highest hardware interrupt level, interrupting any other thread or interrupt handler.

4.2.1 Operating system clock setting (CLK-B-CFG)

The "OS clock setting" test examines the setting of the clock tick period in the operating system. This test shows the default clock timing as they are set by the BSP and/ or the kernel.

Remark that in Compact 7, when performing a sleep(0), the call will immediately return (if no other threads are running at the same priority level). Other sleeps behave normal. Microsoft provides a SleepTillTick() call for this purpose:

Thus in practice:

- Sleep(0) will not sleep, but will yield if there is another thread of same priority runnable.
- SleepTillTick() will sleep between 0-1 ms.
- Sleep(n) will sleep between n and (n-1) ms..

4.2.1.1 Test results

Test	result
Test succeeded	Yes (SleepTillTick)
Tested clock period	1ms
Clock period adaptable	NO

4.2.2 Clock tick processing duration (CLK-P-DUR)

The "clock tick processing duration" test examines the clock tick processing duration in the kernel. The test results are extremely important, as the clock interrupt will disturb all the other performed measurements. Using a tickles kernel will not even prevent this from happening (it will only lower the number of occurrences). The kernel under test was not using the tickles timer option.

The bottom line of the figures in section 4.2.2.2 represents the normal loop time of the test if no clock interrupt occurs during the test loop. The upper line is generated by the samples when a clock interrupt occurred during the loop. The difference between the two lines is the clock tick processing duration.

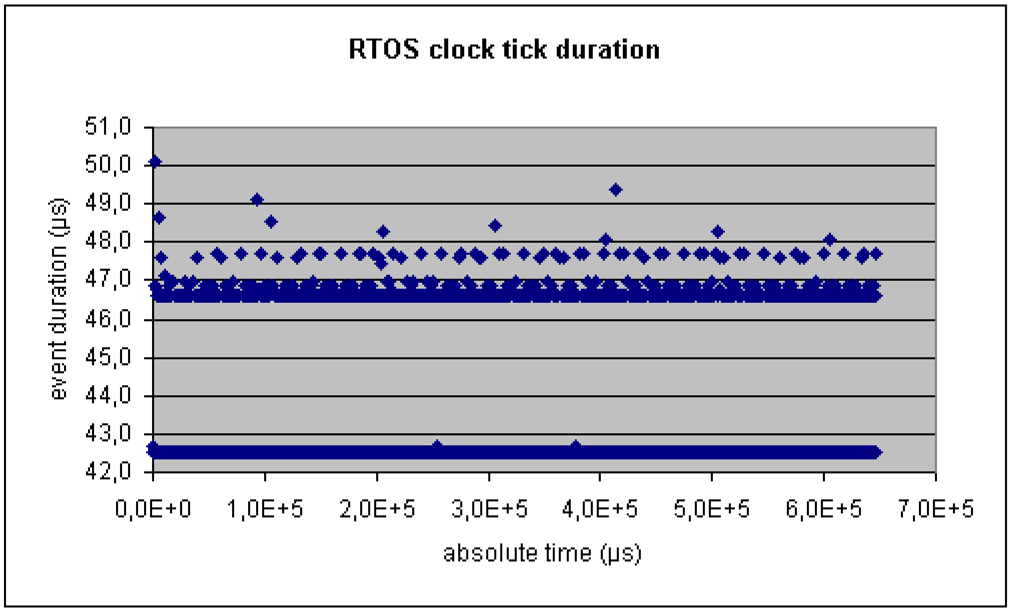
Remark that it is impossible to do not have the clock tick interrupt in the L2 cache! As the L2 cache size is 512KB and 1ms clock tick period is used, the memory bandwidth should be larger than 500MB/s to flush the cache. This is more than double the memory bandwidth available on this platform. As a result, slow down can only be caused by L1 cache misses and thus the clock tick duration is more stable than for instance on a Pentium MMX (our standard evaluation platform).

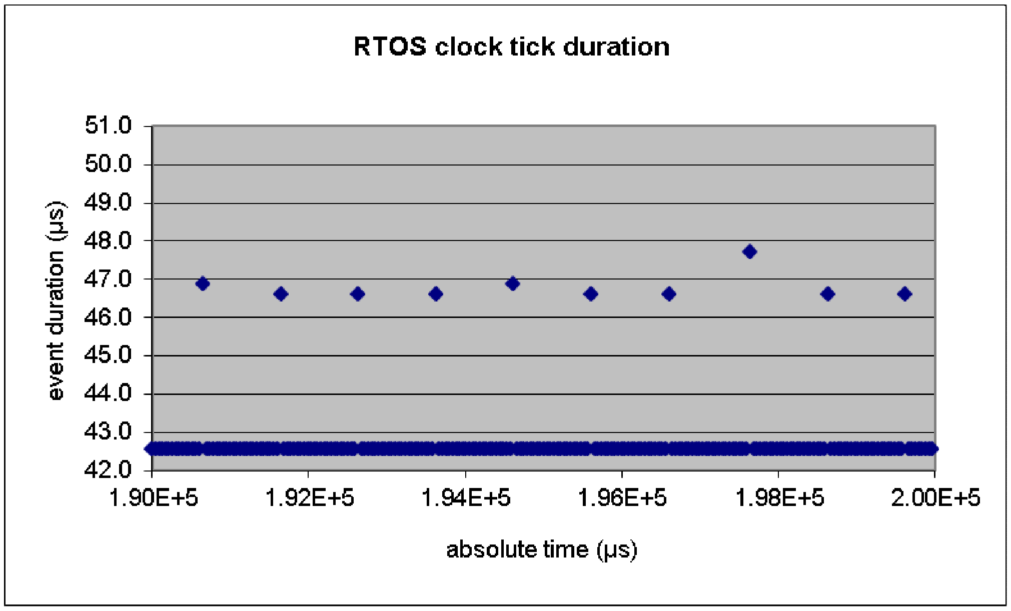
Be careful: the OAL layer may stop the clock tick interrupt if not needed (tickles kernel). However this will only happen if the CPU load is minimal, in which case the used memory bandwidth will be small.

4.2.2.1 Test results

Test	result
CLOCK_LOOP_COUNTER	1000
Normal busy loop time	42.8 µs
Busy loop time with clock interrupt	46.8 µs, worst case 50 µs
Clock interrupt duration	4 µs to 7us

4.2.2.2 Diagrams





Zoomed in extract from diagram above.

4.3 Thread tests (THR)

“Thread tests” measure the scheduler performance.

4.3.1 Thread creation behaviour (THR-B-NEW)

The “thread creation behavior” test examines the OS behavior when it creates threads. This test attempts to answer the question: Does the OS behave as it should in order to be considered a real-time operating system? Following scenarios are tested:

- If a thread is created with a lower priority than the creating thread, then are we sure that it is not activated until the creating thread is finished?
- If a thread is created with the same priority as the creating thread, will it be placed at the ready tail?
- When yielding after the creation in the above test, does the newly created thread becomes active?
- If a thread is created with a higher priority than the creating thread, is it then immediately activated?

This test succeeded without any problems.

4.3.1.1 Test results

Test	result
Test succeeded	YES
Lower priority not activated?	YES
Same priority at tail?	YES
Yielding works?	YES
Higher priority activated?	YES

4.3.2 Round robin behaviour (THR-B-RR)

The “round robin behavior” test checks if the scheduler uses a fair round robin mechanism to schedule threads that use the SCHED_RR scheduling policy, are of the same priority, and are in the ready-to-run state (and using)!

☹ A problem was detected here: the first time a thread becomes active, it takes a longer time slice (100ms) than in the other cases (1ms = clock tick).

We took a look back to the test results of **CE 6.0** and discovered that the same problem was indeed present there as well. To be sure that we didn’t do anything wrong in the test, we compared the code for this test with the test code of other RTOS which did not have this behavior and no differences were found.

Although it is strange behavior, creating dynamically the threads in a real-time system is a bad practice which is normally never done. As such, this problem will not have any impacts in real use cases.

4.3.2.1 Test results

Test	result
Test succeeded	No (first time slice after thread creation is longer)
RR Time slice following this test	1 clock tick normally (first slice takes 100 clock ticks)

4.3.3 Thread switch latency between same priority threads (THR-P-SLS)

The “thread switch latency between same priority threads” test measures the time needed to switch between threads of the same priority. For this test, threads must voluntarily yield the processor for other threads.

In this test, we use the SCHED_FIFO policy. If we do not use the “first in first out” policy, a round-robin clock event could occur between the yield and the trace, so that the thread activation is not seen in the trace.

This test was performed in order to generate the worst-case behavior. We performed the test with an increasing number of threads, starting with two (2) and going up to 1000 in order to observe the behavior in a worst-case scenario. As we increase the number of active threads, the caching effect becomes evident since the thread context will no longer be able to reside in the cache (on this platform the L1 caches are 32KB, both for the data as the instruction cache).

As loading/starting the test software passes a lot of code and data to the processor, the first clock tick will not be cached in L2 (causing the peak for the first clock tick in the 2 thread scenarios). Once there are enough running threads, the clock interrupt will be always un-cached and thus from the 128 thread tests, the clock interrupts always generate a delay of approximately 7µs.

The thread switch latency is longer compared with its best competitors (about a factor 2).

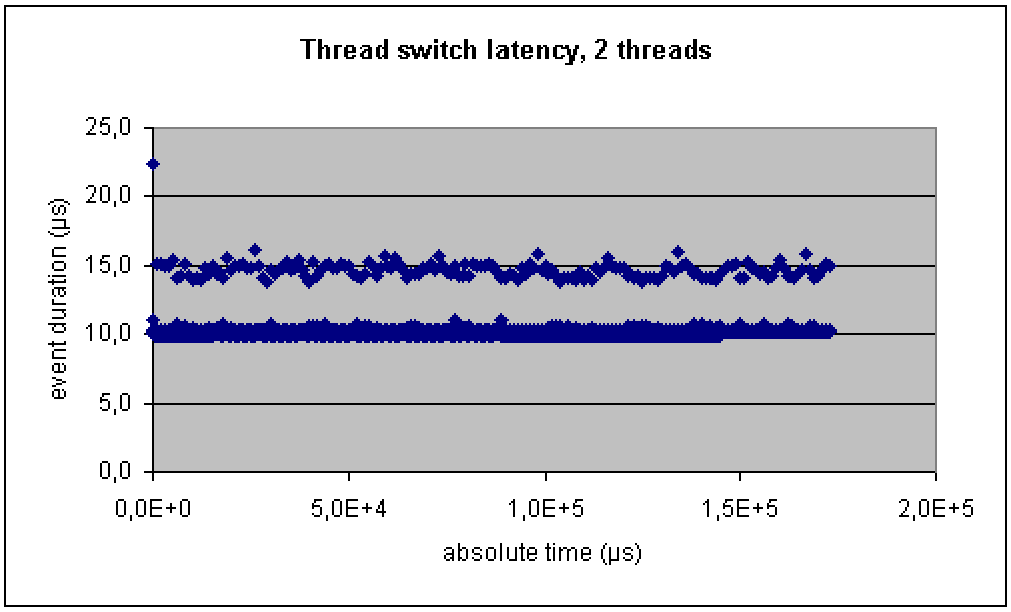
4.3.3.1 Test results

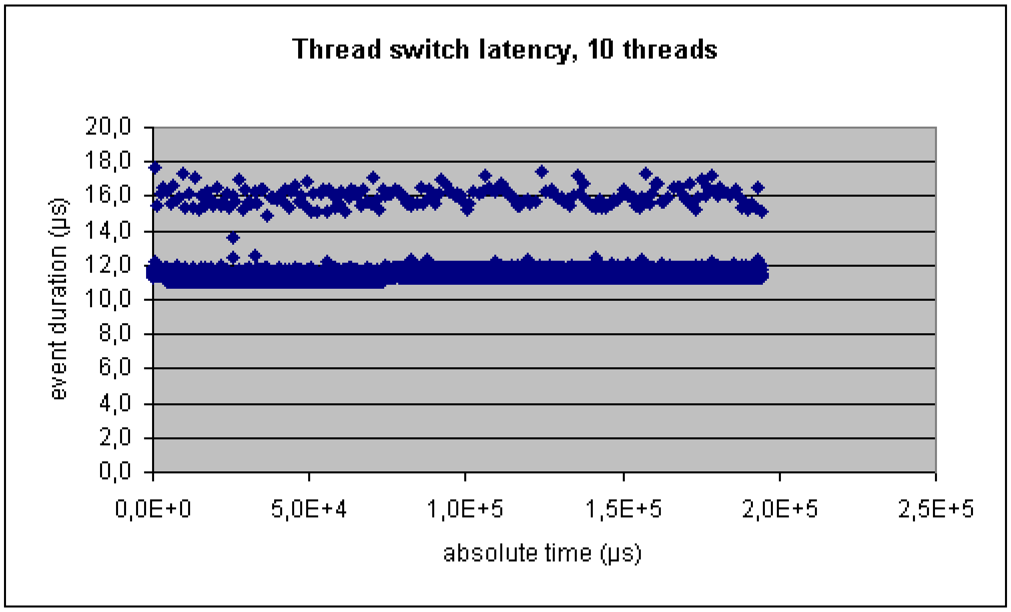
Test	result

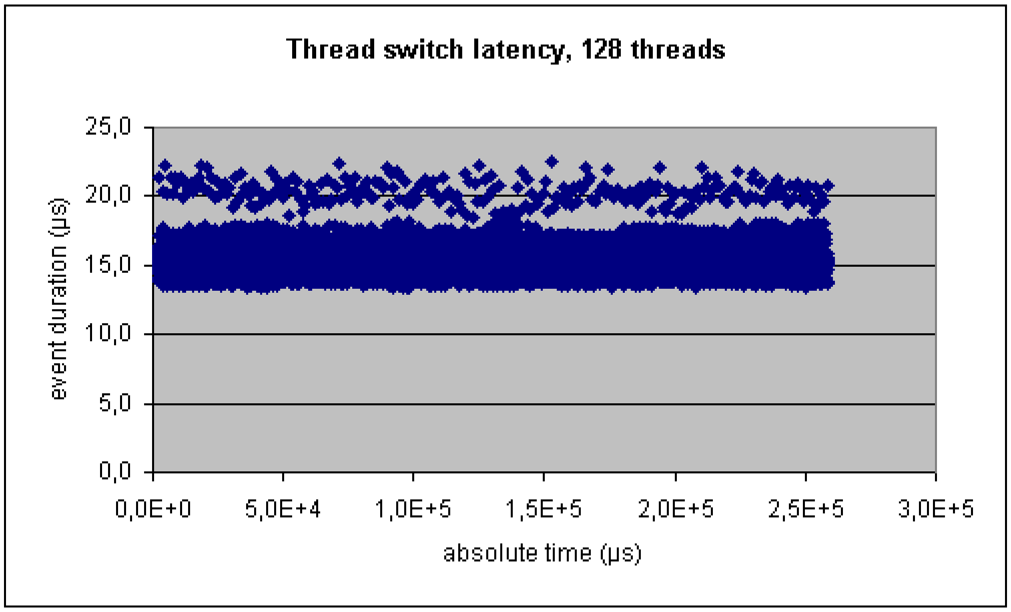
Test	result
Test succeeded	YES

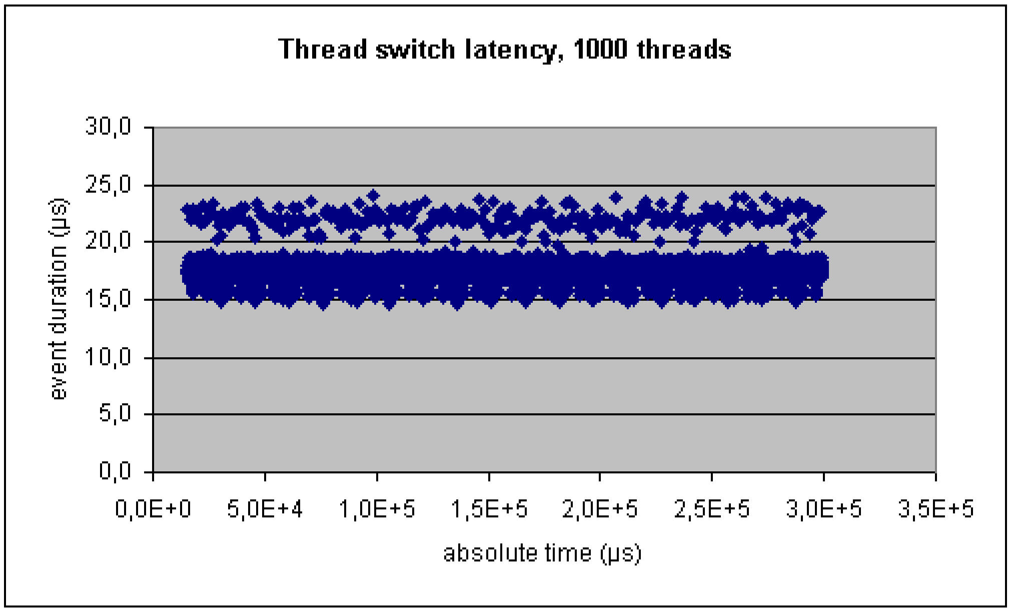
Test	Sample qty	Avg	Max	Min
Thread switch latency, 2 threads	16383	10.1 µs	22.3 µs	9.9 µs
Thread switch latency, 10 threads	16379	11.4 µs	17.6 µs	11.0 µs
Thread switch latency, 128 threads	16320	15.3 µs	22.5 µs	13.3 µs
Thread switch latency, 1000 threads	15884	17.4 µs	24.1 µs	14.7 µs

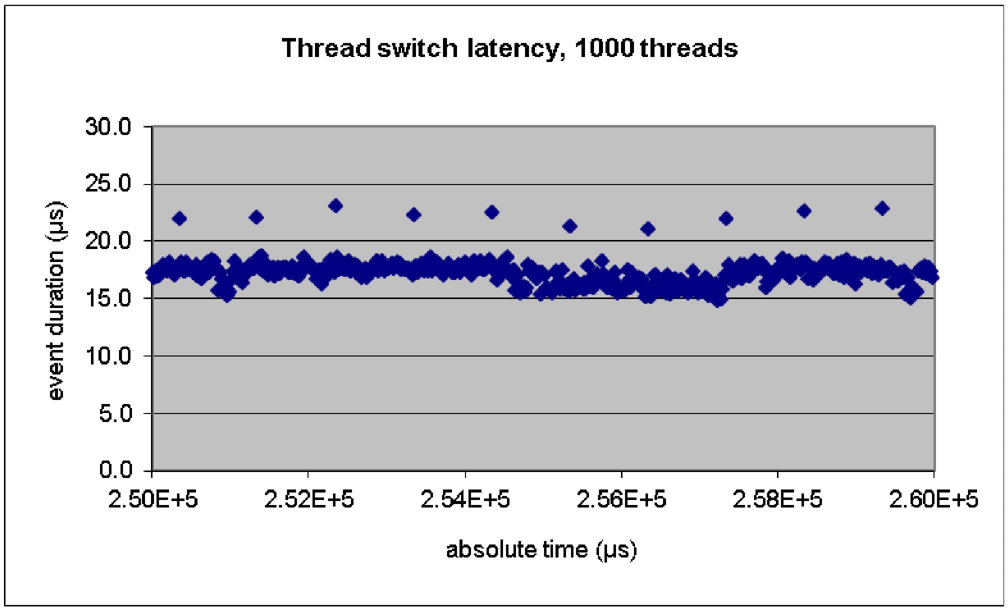
4.3.3.2 Diagrams




Thread switch latency, 10 threads
event duration (µs)
20,0
18,0
16,0
14,0
12,0
10,0
8,0
6,0
4,0
2,0
0,0
0,0E+0
5,0E+4
1,0E+5
1,5E+5
2,0E+5
2,5E+5
absolute time (µs)


Thread switch latency, 128 threads
event duration (µs)
25,0
20,0
15,0
10,0
5,0
0,0
0,0E+0
5,0E+4
1,0E+5
1,5E+5
2,0E+5
2,5E+5
3,0E+5
absolute time (µs)





Zoomed extract from 1000 threads test.

4.3.4 Thread creation and deletion time (THR-P-NEW)

The “thread creation and deletion time” test examines the time required to create a thread, and the time required to delete a thread in the following different scenarios:

- Scenario 1 “never run”: The created thread has a lower priority than the creating thread and is deleted before it has any chance to run. No thread switch occurs in this test.
- Scenario 2 “run and terminate”: The created thread has a higher priority than the creating thread and will be activated. The created thread immediately terminates itself (thread does nothing).
- Scenario 3 “run and block”: The same as the previous scenario (scenario 2: run and terminate), but the created thread does not terminate (it lowers its priority when it is activated).

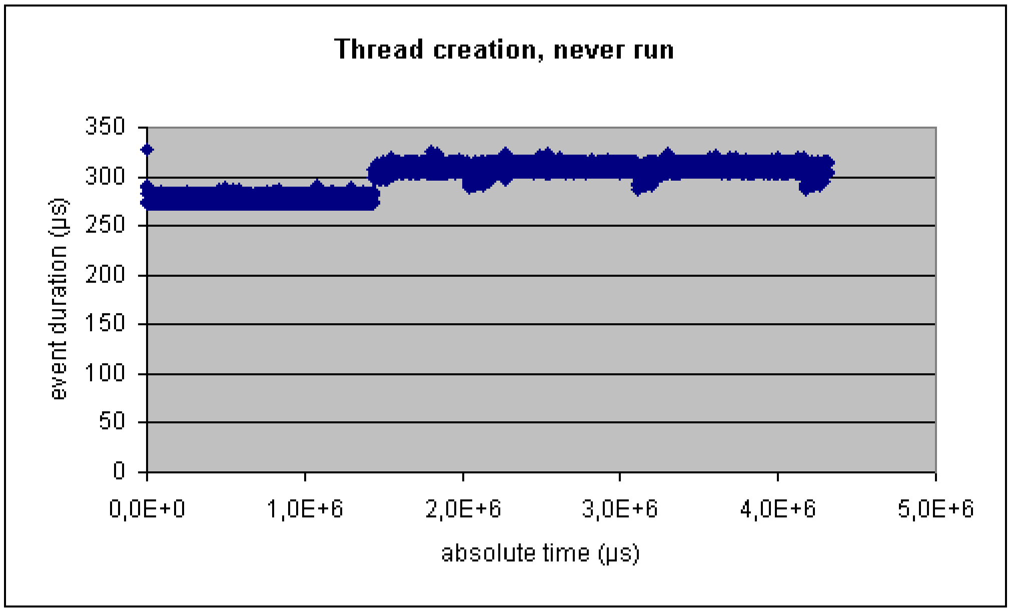
In the scenarios where the thread actually runs (2 and 3), the creation time is measured as the duration from the system call creating the thread until the time when the created thread is activated. For the “never run” scenario, the creation time is measured as the duration of the system call.

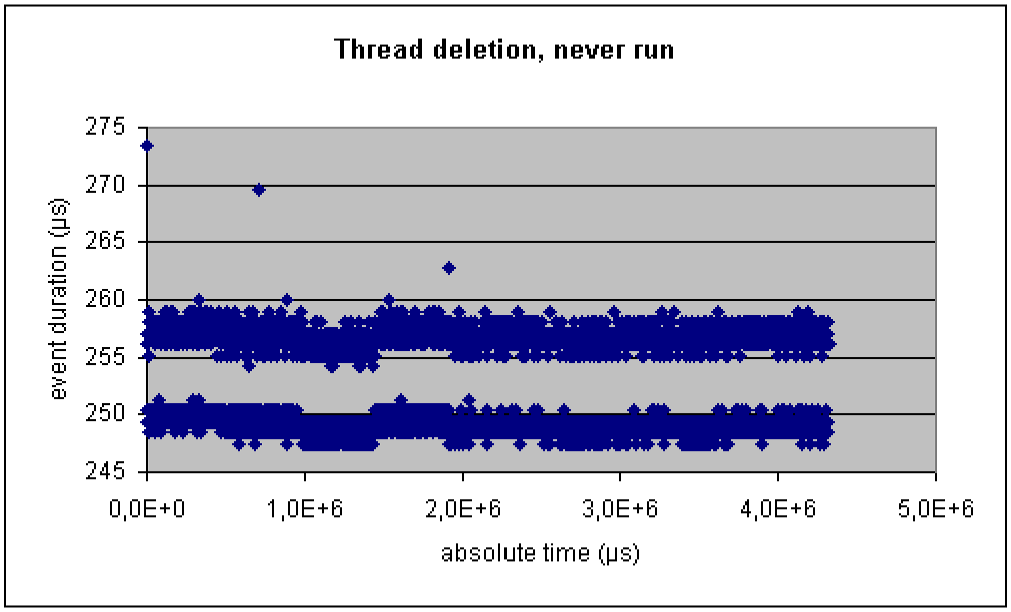
4.3.4.1 Test results

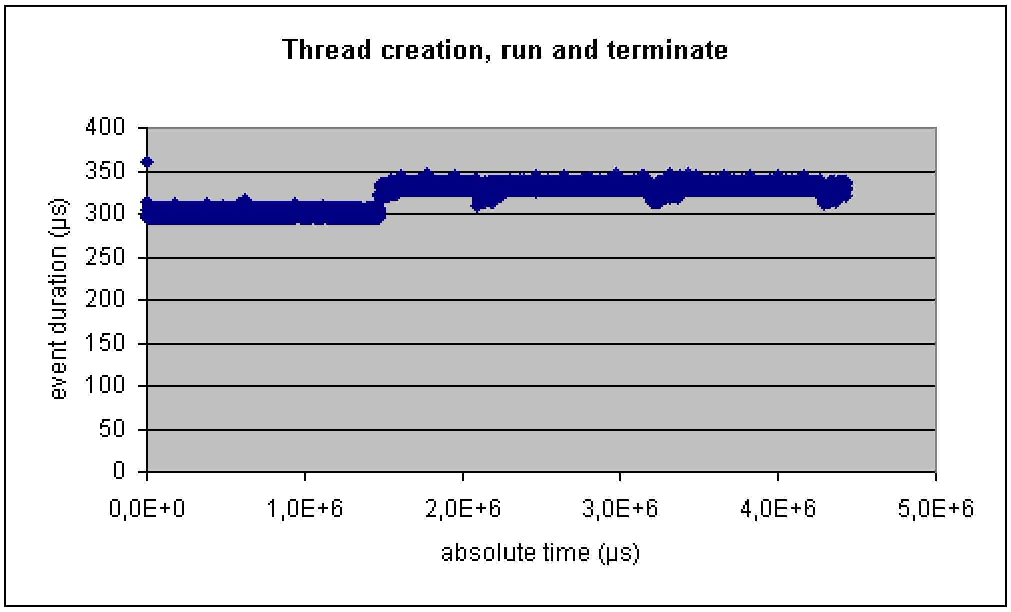
Test	result
Test succeeded	YES

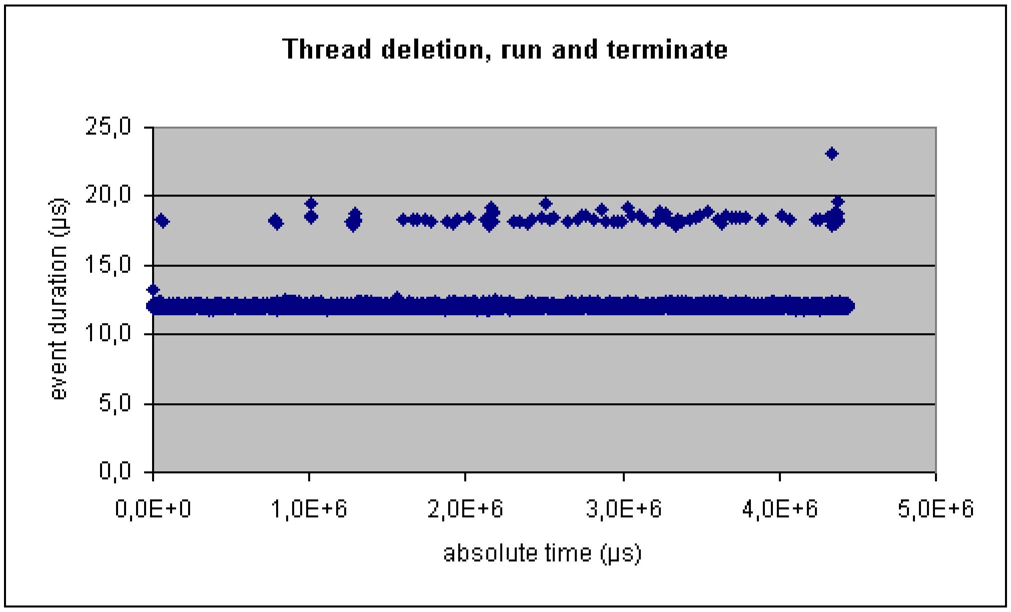
Test	Sample qty	Avg	Max	Min
Thread creation, never run	4096	296 µs	328 µs	273 µs
Thread deletion, never run	4095	251 µs	273 µs	248 µs
Thread creation, run and terminate	7500	320 µs	361 µs	296 µs
Thread deletion, run and terminate	7500	12.1 µs	23.1 µs	11.8 µs
Thread creation, run and block	7500	321 µs	367 µs	297 µs
Thread deletion, run and block	7500	254 µs	278 µs	250 µs

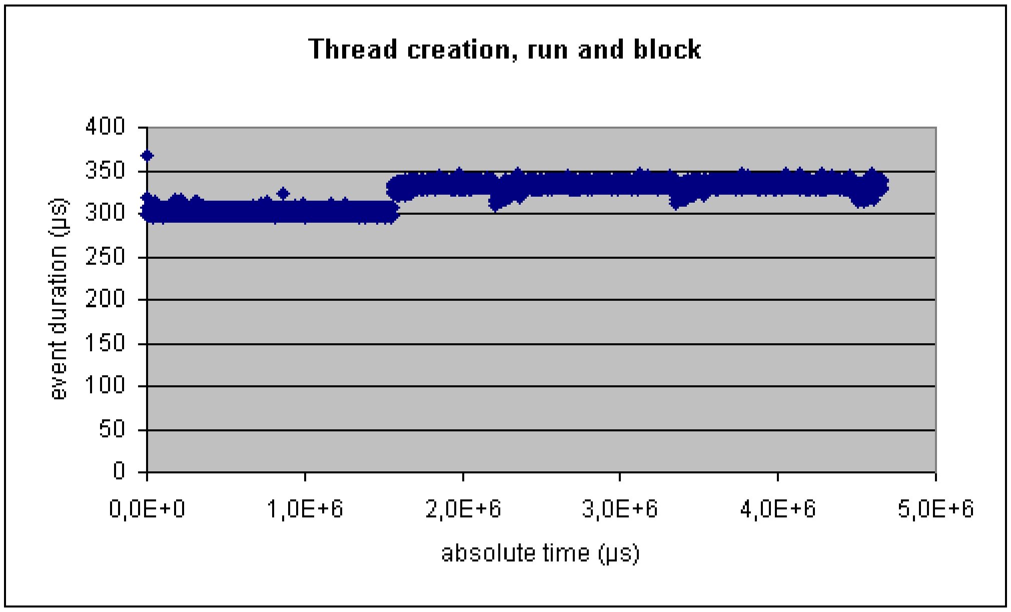
4.3.4.2 Diagrams

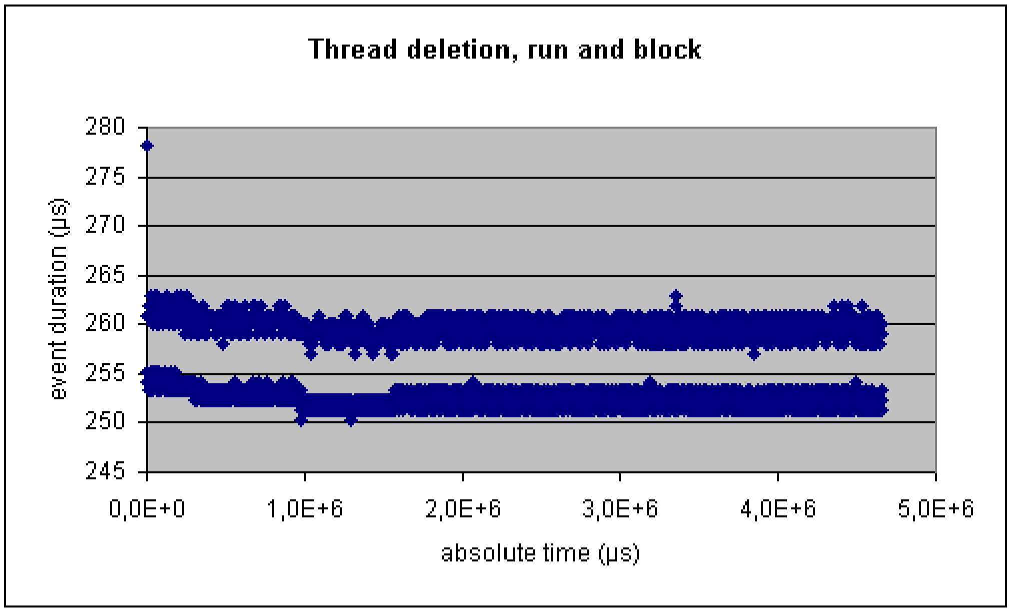

Thread creation, never run
event duration (µs)
350
300
250
200
150
100
50
0
0,0E+0
1,0E+6
2,0E+6
3,0E+6
4,0E+6
5,0E+6
absolute time (µs)


Thread deletion, never run
event duration (µs)
275
270
265
260
255
250
245
0,0E+0
1,0E+6
2,0E+6
3,0E+6
4,0E+6
5,0E+6
absolute time (µs)


Thread creation, run and terminate
event duration (µs)
400
350
300
250
200
150
100
50
0
0,0E+0
1,0E+6
2,0E+6
3,0E+6
4,0E+6
5,0E+6
absolute time (µs)


Thread deletion, run and terminate
event duration (µs)
25,0
20,0
15,0
10,0
5,0
0,0
0,0E+0
1,0E+6
2,0E+6
3,0E+6
4,0E+6
5,0E+6
absolute time (µs)





4.4 Semaphore tests (SEM)

“Semaphore tests” examine the behavior and performance of the OS counting semaphore. The counting semaphore is a system object that can be used to synchronize threads.

4.4.1 Semaphore locking test mechanism (SEM-B-LCK)

In this test, we verify if the counting semaphore locking mechanism works as it is expected to work. If this mechanism works as expected, then:

- The P () call will block only when the count is zero.
- The V () call will increment the semaphore counter.
- In the case where the semaphore counter is zero, the V () call will cause a rescheduling by the OS, and blocked threads may become active.

The semaphore behaves correctly as a protection mechanism.

4.4.1.1 Test results

Test	result
Test succeeded	YES
Maximum semaphore value?	Limited by the "int" type
Rescheduling on free?	OK

4.4.2 Semaphore releasing mechanism (SEM-B-REL)

The "semaphore releasing mechanism" test verifies that the highest priority thread being blocked on a semaphore will be released by the release operation. This action should be independent of the order of the acquisitions taking place.

Compact 7 passed this test.

4.4.2.1 Test results

Test	result
Test succeeded	YES

4.4.3 Time needed to create and delete a semaphore (SEM-P-NEW)

The "time needed to create and delete a semaphore" test is performed to gain an insight about the time needed to create a semaphore and the time needed to delete it. The deletion time is checked in two cases:

- The ***semaphore*** is used between the creation and deletion.
- The ***semaphore*** is NOT used between the creation and deletion.

Remark that although we do not use "named" ***semaphores***, there seems to be a system call required to create/delete a semaphore.

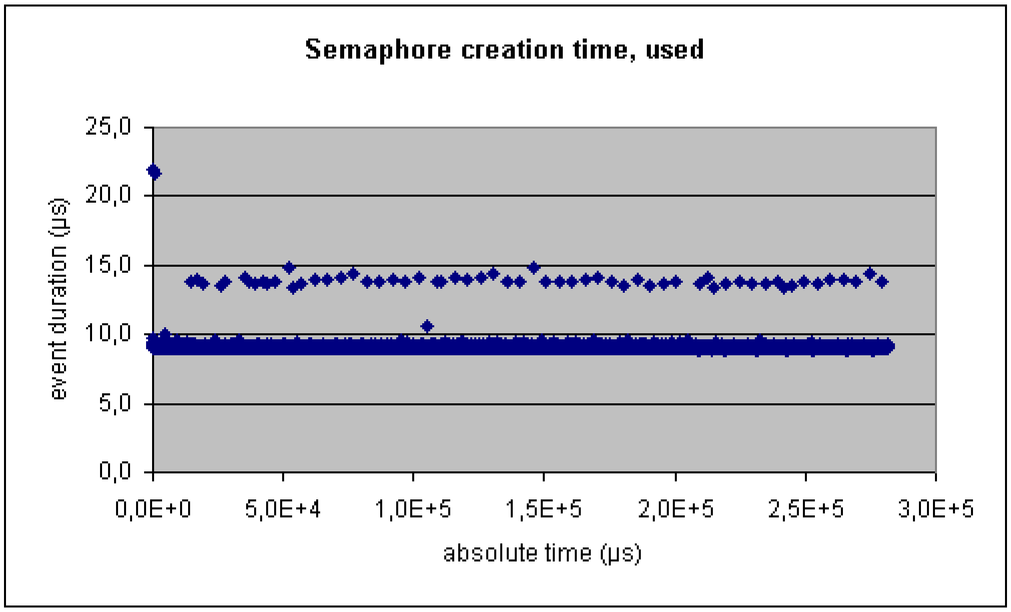
The clock tick line is clearly visible (Diagrams of section 4.4.3.2). On some diagrams, you can see that the clock interrupt starts to take a bit longer once there is more than 100ms full CPU load.

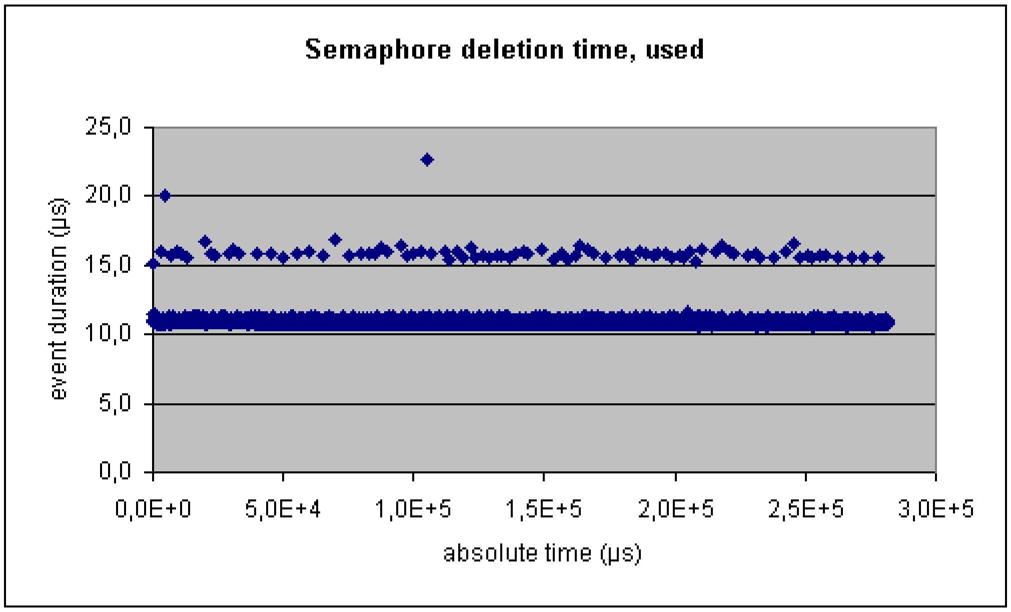
4.4.3.1 Test results

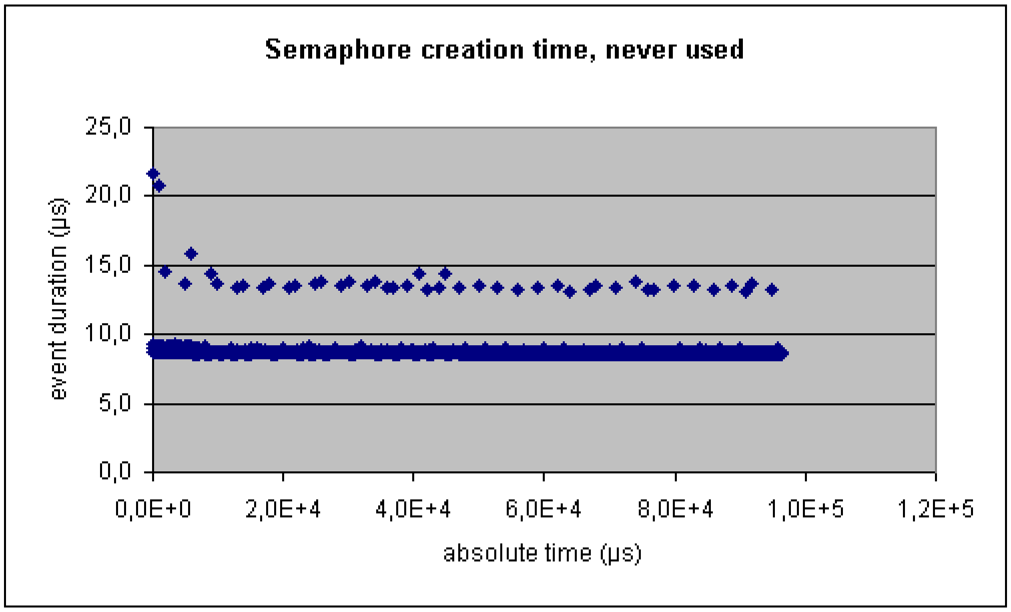
Test	result
Test succeeded	YES

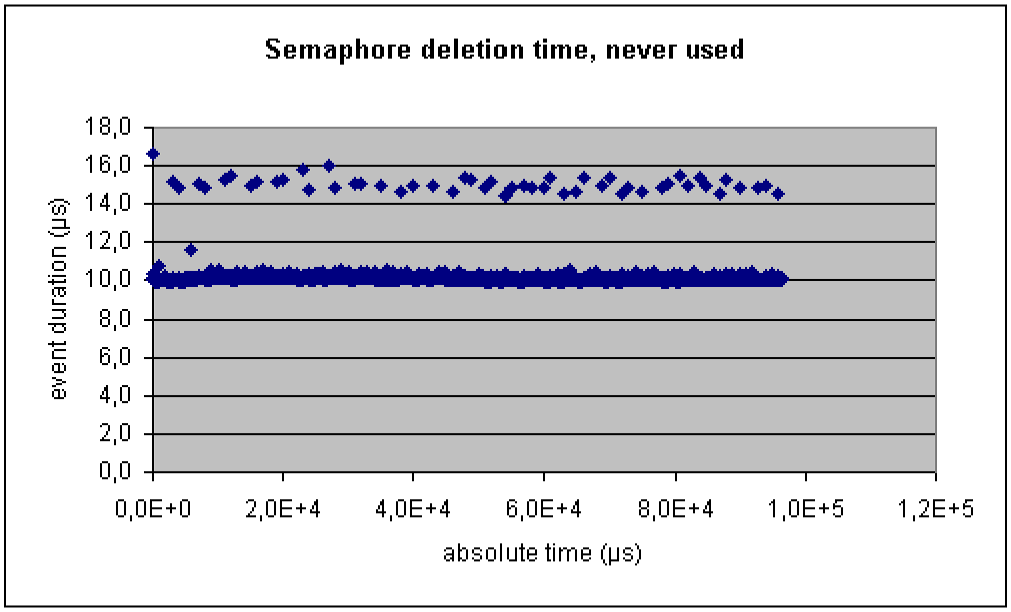
Test	Sample qty	Avg	Max	Min
Semaphore creation time, used	7500	9.1 µs	21.9 µs	8.9 µs
Semaphore deletion time, used	7500	10.9 µs	22.6 µs	10.7 µs
Semaphore creation time, never used	7500	8.8 µs	21.6 µs	8.6 µs
Semaphore deletion time, never used	7500	10.2 µs	16.6 µs	9.9 µs

4.4.3.2 Diagrams




Semaphore deletion time, used
event duration (µs)
25,0
20,0
15,0
10,0
5,0
0,0
0,0E+0
5,0E+4
1,0E+5
1,5E+5
2,0E+5
2,5E+5
3,0E+5
absolute time (µs)


Semaphore creation time, never used
event duration (µs)
25,0
20,0
15,0
10,0
5,0
0,0
0,0E+0
2,0E+4
4,0E+4
6,0E+4
8,0E+4
1,0E+5
1,2E+5
absolute time (µs)



4.4.4 Test acquire-release timings: non-contention case (SEM-P-ARN)

The “acquire-release timings: non-contention case” test measures the acquisition and release time in the non-contention case. Since in this test the semaphore does not neither block nor causes any rescheduling (thread switching), the duration of the call should be short.

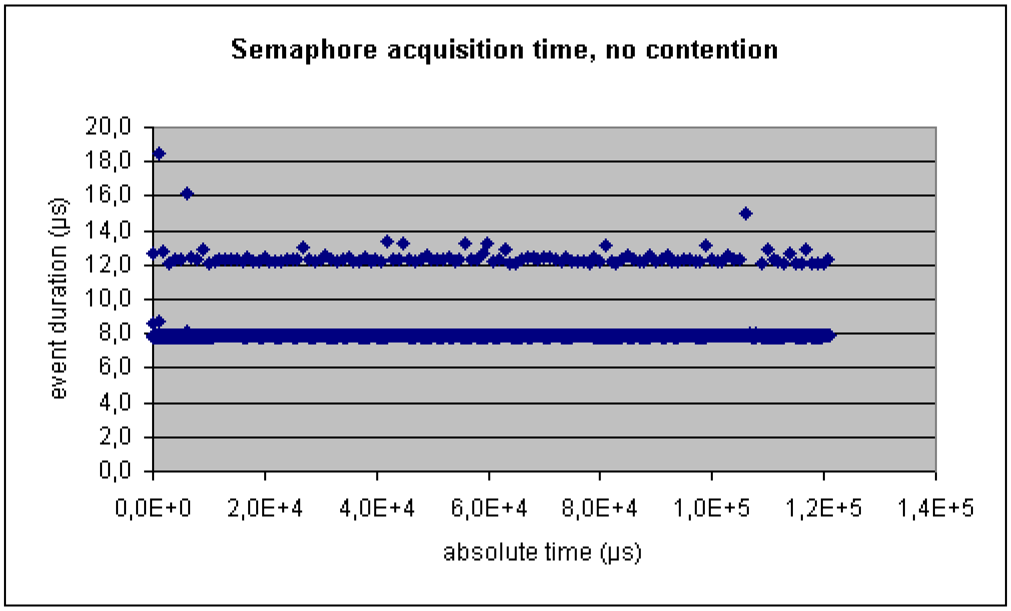
A system call is used even that the ***semaphore*** is not contended.

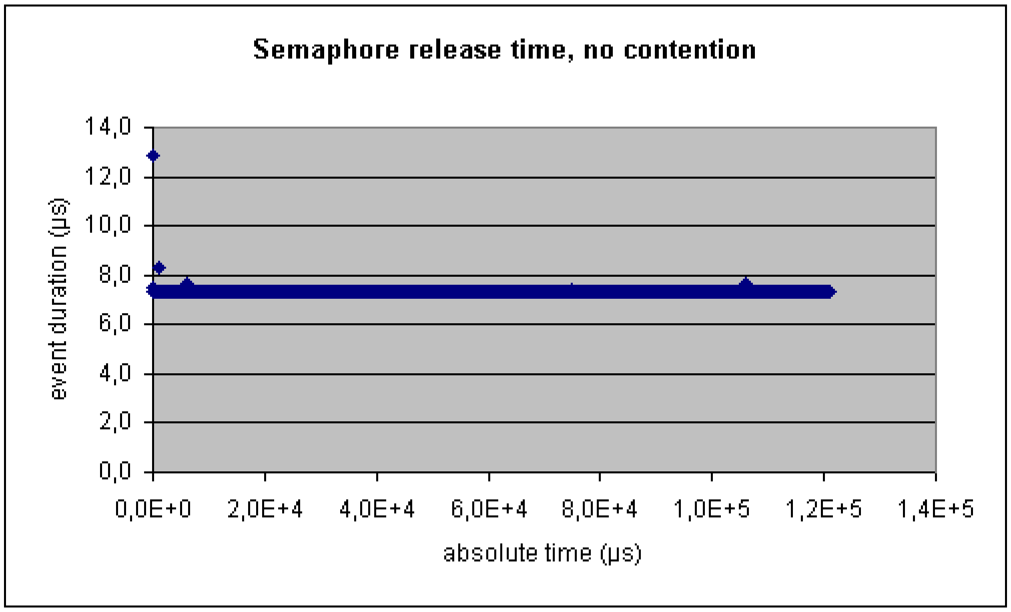
4.4.4.1 Test results

Test	result
Test succeeded	YES

Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, no contention	7500	8.0 µs	18.5 µs	7.8 µs
Semaphore release time, no contention	7500	7.3 µs	12.8 µs	7.3 µs

4.4.4.2 Diagrams





4.4.5 Test acquire-release timings: contention case (SEM-P-ARC)

The “acquire release timings: contention case” test is performed to test the time needed to acquire and release a semaphore, depending on the number of threads blocked on the semaphore. It measures the time in the contention case when the acquisition and release system call causes a rescheduling to occur.

The purpose of this test is to see if the number of blocked threads has an impact on the times needed to acquire and release a semaphore. It attempts to answer the question: "How much time does the OS needs to find out which thread should be scheduled first?"

In this test, since each thread has a different priority, the question is how the OS handles these pending thread priorities on a semaphore. To have a more clear view on our test, you can take a look on the expanded diagrams during a small time frame (e.g. one test loop):

- We create 128 threads with different priorities. The creating thread has a lower priority than the threads being created.
- When the thread starts execution, it tries to acquire the ***semaphore***; but as it is taken, the thread stops and the kernel switch back to the creating thread. The time from the acquisition attempt (which fails) to the moment the creating thread is activated again is called here the "acquisition time". Thus, this time includes the thread switch time.
- Thread creation takes some time, so the time between each measurement point is large compared with most other tests.
- After the last thread is created and is blocked on the ***semaphore***, the creating thread starts to release the ***semaphore*** repeating this action the same number of times as the number of blocked threads on the semaphore.
- We start timing at the moment the ***semaphore*** is released which in turn will activate the pending thread with the highest priority, which will stop the timing (thus again the thread switch time is included).

Now, the most important part of this test is to see if the number of threads pending on a ***semaphore*** has an impact on release times. Clearly, it doesn't, so this is good.

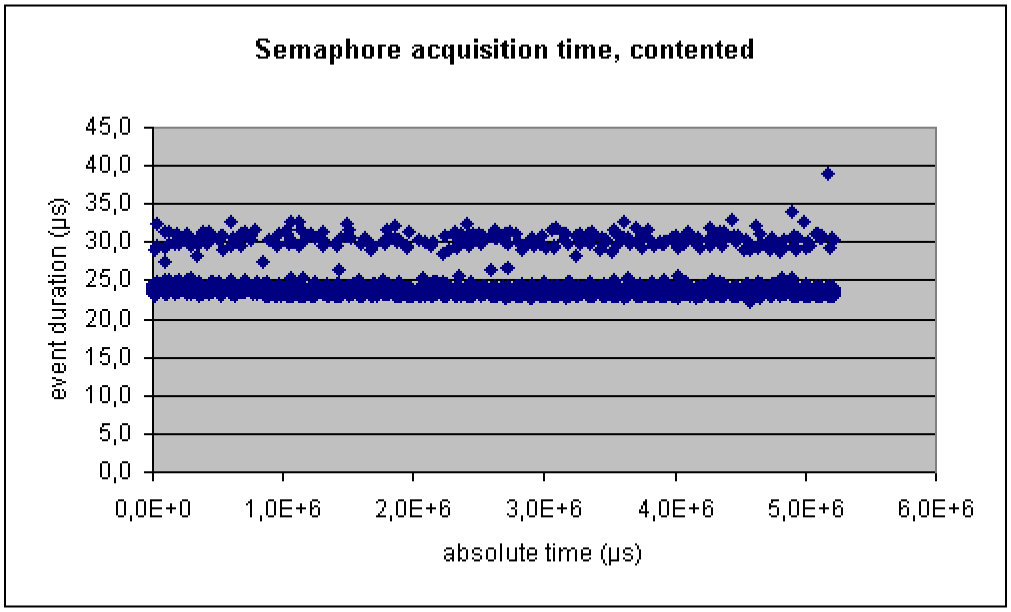
There is some impact on the release, but probably this is more related with caching effect.

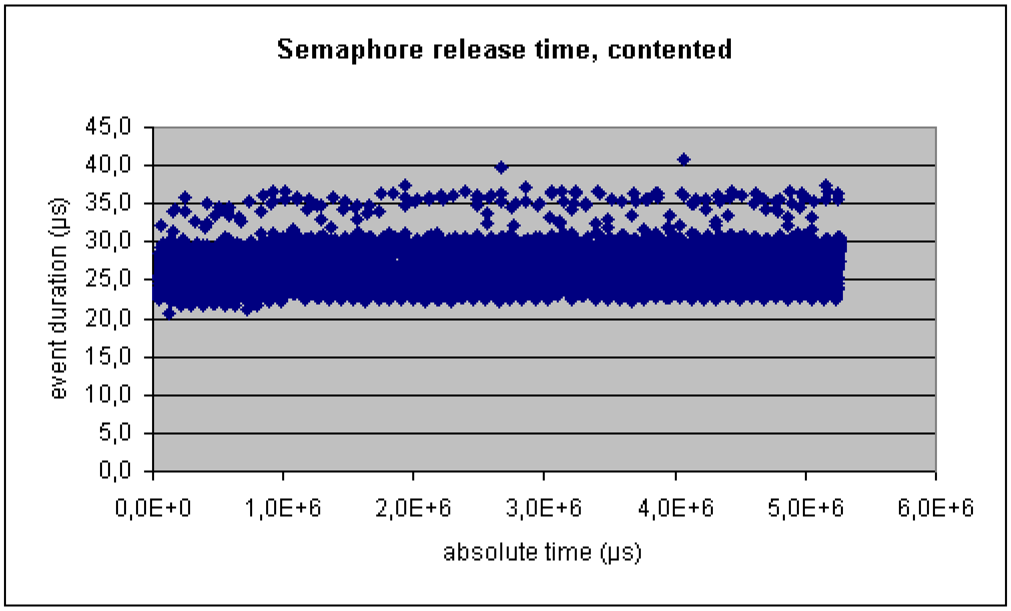
4.4.5.1 Test results

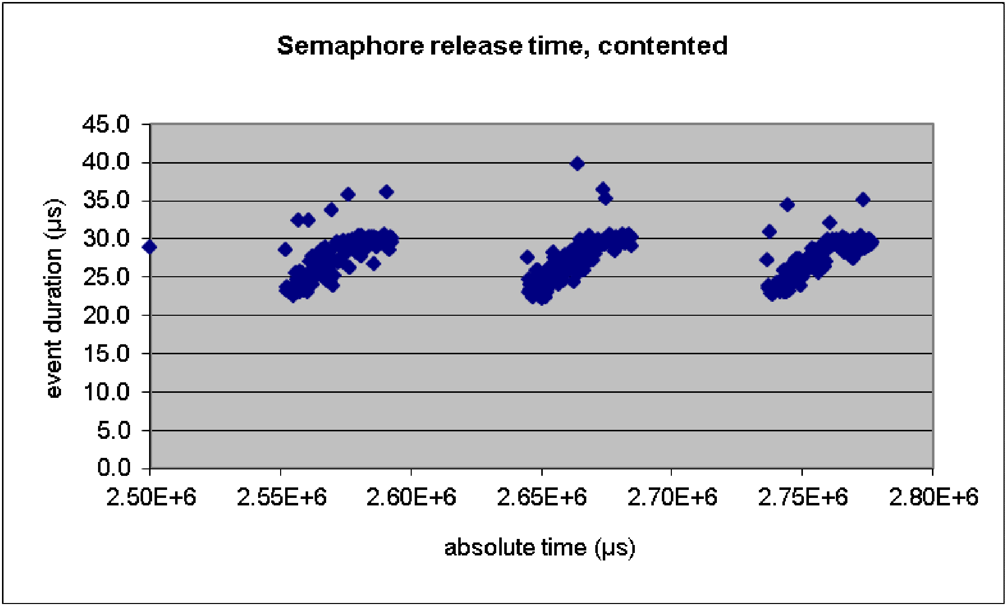
Test	result
Test succeeded	YES
Max number of threads pending	128

Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, contented	7424	23.8 µs	38.9 µs	22.2 µs
Semaphore release time, contented	7424	27.2 µs	40.7 µs	20.6 µs

4.4.5.2 Diagrams







Zoomed in version of previous diagram.

4.5 Mutex tests (MUT)

Our "mutex tests" help us evaluate the behavior and performance of the mutual exclusive semaphore.

Although the mutual exclusive semaphore (further called mutex) is usually described as being the same as a counting semaphore where the count is one, this is not true. The behavior of a mutex is completely different than the behavior of a semaphore. Unlike semaphores, mutexes use the concept of a "lock owner", and can thus be used to prevent priority inversions. Semaphores cannot do this, and it goes without saying that mutexes (and not semaphores) should not be used semaphores for critical section protection mechanisms. In scope of the framework, this test will look into detail of a mutex system object that avoids priority inversion.

Remark that, Compact 7 has as well a ***Mutex*** system object; but this should be used only between processes as it always requires a long round-trip to the kernel even if the lock is not contented.

Remark as well that there exists InterlockedXXX functions, which use the available CPU instruction set to provide atomic behavior and as a result, these are fast.

4.5.1 Priority inversion avoidance mechanism (MUT-B-ARC)

The “priority inversion avoidance mechanism” test determines if the system call being tested prevents the priority inversion case. To check this possibility, the test artificially creates a priority inversion.

Priority inversion behaves as expected.

4.5.1.1 Test results

Test	result
Priority inversion avoidance system call present	Yes
System call used	InitializeCriticalSection, EnterCriticalSection, LeaveCriticalSection
Test succeeded	YES
Priority inversion avoided	YES
Mechanism used if any?	Priority inversion cannot be disabled in **Compact 7**, which is a plus!

4.5.2 Mutex acquire-release timings: contention case (MUT-P-ARC)

The “mutex acquire-release timings: contention case” test is the same test as the “priority inversion avoidance mechanism” test described above, but performed in a loop. In this case, we measure the time needed to acquire and release the mutex in the priority inversion case.

Our test is designed so that the acquisition enforces a thread switch:

- The acquiring thread is blocked
- The thread with the lock is released.

We measured the acquisition time from the request for the mutex acquisition to the activation of the lower priority thread with the lock.

Note that before the release, an intermediate priority level thread is activated (between the low priority one having the lock and the high priority one asking the lock). Due to the priority inheritance, this thread does not start to run (the low priority thread having the lock inherited the high priority of the thread asking the lock).

We measured the release time from the release call to the moment the thread requesting the mutex was activated; so this measurement also includes a thread switch.

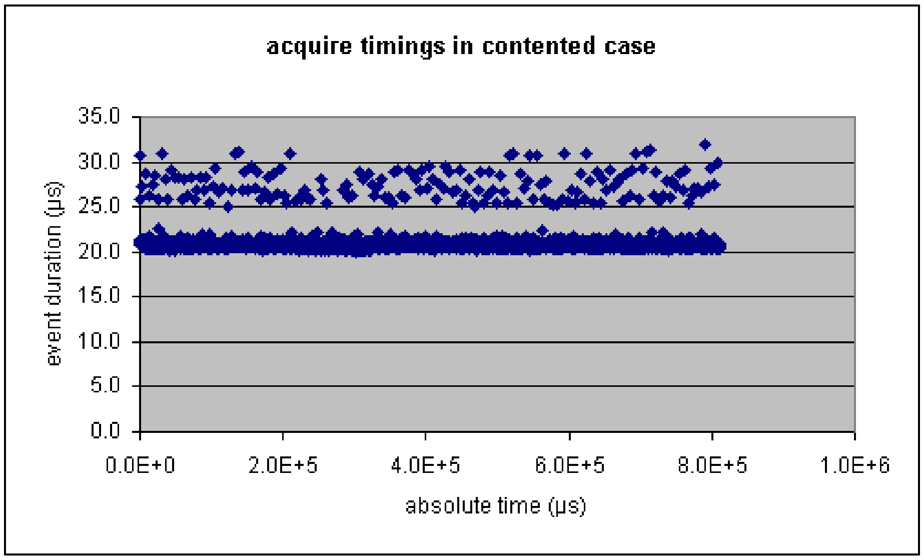
The clock tick interrupt can be clearly seen (as usual) (figures of section 4.5.2.2).

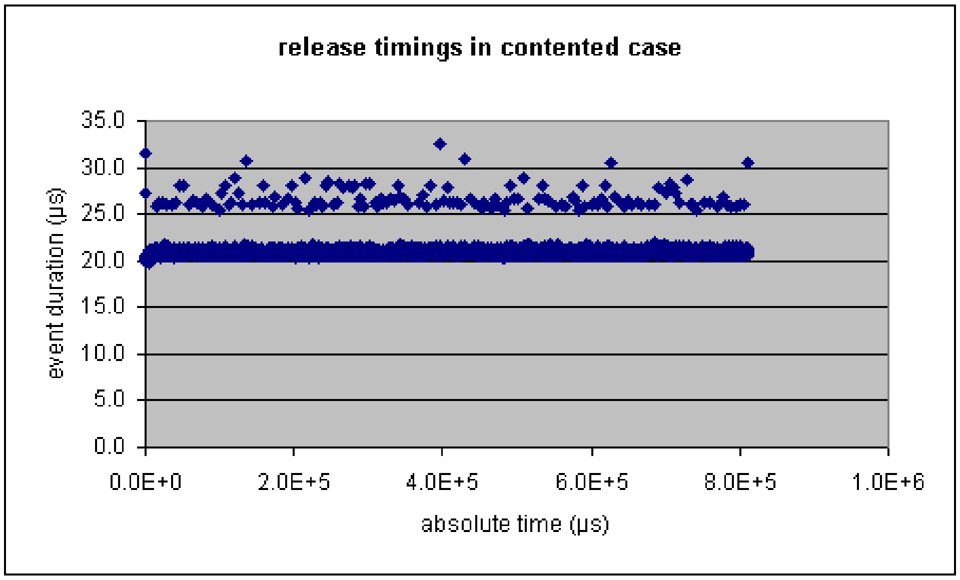
4.5.2.1 Test results

Test	result
Test succeeded	Yes

Test	Sample qty	Avg	Max	Min
Mutex acquisition time, contention	7500	20.9 µs	31.9 µs	20.0 µs
Mutex release time, contention	7500	20.1 µs	32.6 µs	19.8 µs

4.5.2.2 Diagrams:





4.5.3 Mutex acquire-release timings: non-contention case (MUT-P-ARN)

The "mutex acquire-release timings: no contention case" test measures the overhead incurred by using a lock when this lock is not owned by any other thread. Well-designed software will use non-contended locks most of the time, and only in some rare cases the lock will be taken by another thread.

Therefore, it is important that the non-contention case should be fast. Remark that this is only possible if the CPU supports some type of atomic instruction, so that no system call is needed when no contention is detected.

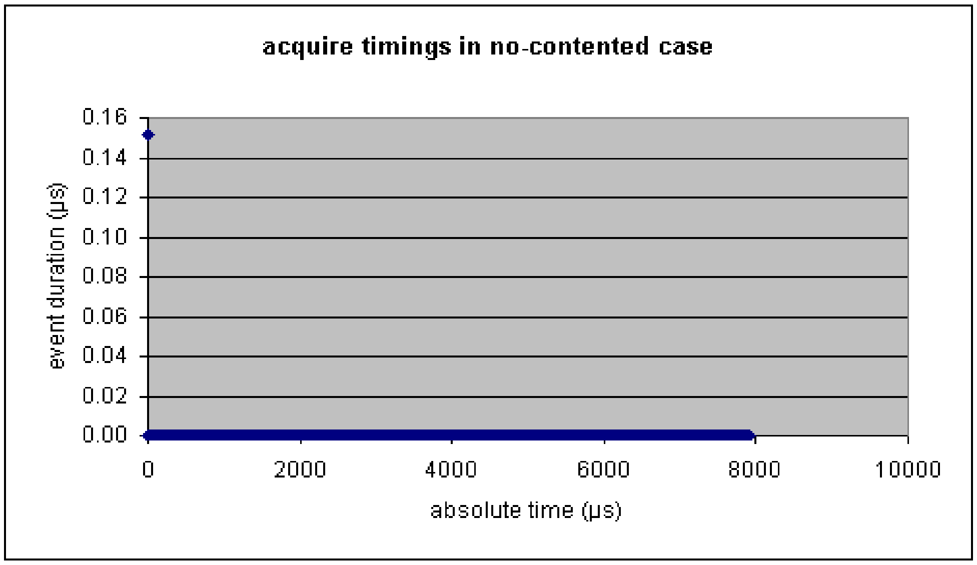
Compact 7 does indeed avoid kernel round trips if you use the CriticalSection system object. This is not the case for the mutex system object, so you should only use the mutex system object for locking between processes (that's the reason why a round-trip to the kernel is needed in this case).

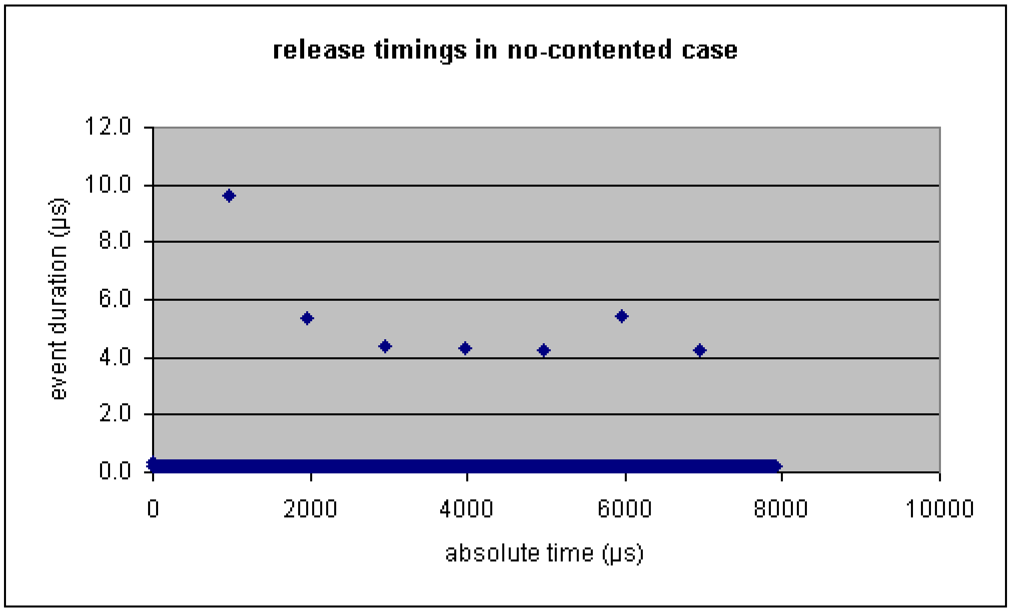
4.5.3.1 Test results

Test	result
Test succeeded	Yes

Test	Sample qty	Avg	Max	Min
Mutex acquisition time, no contention	7500	<0.1 µs	0.2 µs	<0.1 µs
Mutex release time, no contention	7500	0.2 µs	9.6 µs	0.2 µs

4.5.3.2 Diagrams:





4.6 Interrupt tests (IRQ)

"Interrupt tests" evaluate how the operating system performs when handling interrupts.

Interrupt handling is a key system capability of real-time operating systems. Indeed, RTOSs are typically event driven.

For these tests, our standard tracing system is adapted. Interrupts are generated by a plugged-in PCI related card (can be PMC/PCI or CPCI). This card has a complete independent processor on board, with custom-made software. As such, we can guarantee that an independent interrupt source is not synchronized in any way with the platform under test.

4.6.1 Interrupt latency (IRQ_P_LAT)

The "interrupt latency" test measures the time it takes to switch from a running thread to an interrupt handler. This time is measured from the moment the running thread is interrupted, so the measurement does not take into account the hardware interrupt latency.

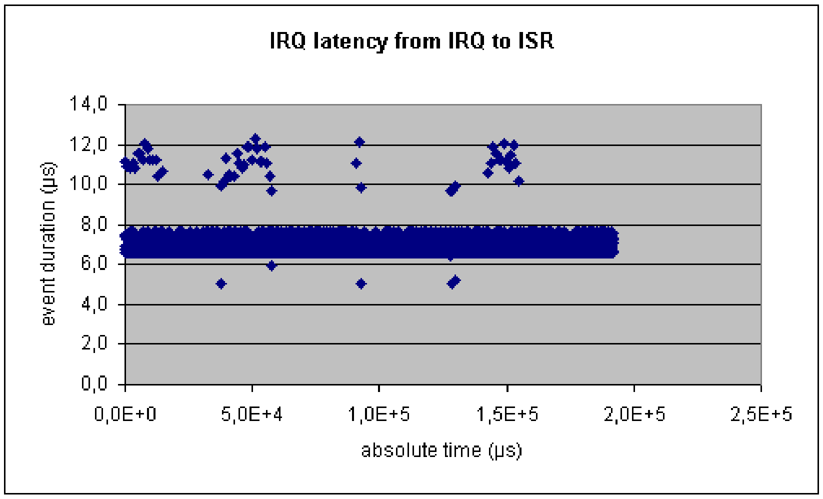
The PCI interrupt line is measured by an analyzer, but this can take only samples if there is traffic on the bus. Therefore, there is a load thread generating PCI traffic. If this load thread is delayed, then it can be that the PCI interrupt pin is detected only a bit later, which is shown the figure below (section 4.6.1.2) as the dots that are lower than 6.2 µs.

The clock time is easily detected again (it has the highest interrupt level).

4.6.1.1 Test results

Test	Sample qty	Avg	Max	Min
Interrupt dispatch latency	6731	6.8 µs	12.3 µs	6.2 µs

4.6.1.2 Diagrams



4.6.2 Interrupt to thread latency (IRQ_P_TLT)

This test measures the time it takes to switch from the interrupt handler to the thread that is activated from the interrupt handler.

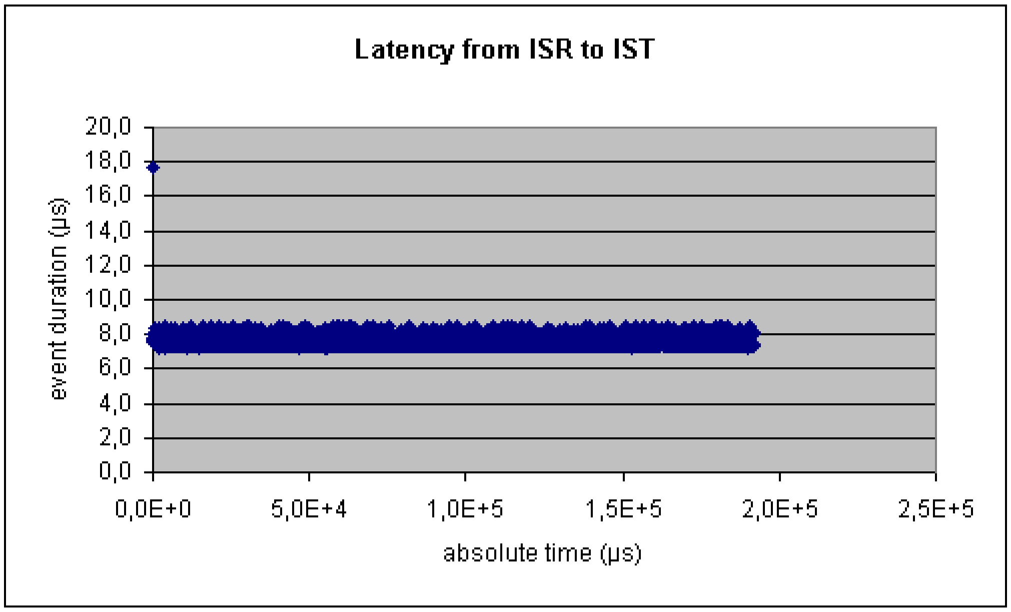
The default interrupt handling in **Compact 7** is to use an ***Interrupt Service Thread (IST***). So the total latency will be the interrupt latency to the interrupt handler together with the interrupt to IST latency.

Thus the total IST latency is around 15us.

4.6.2.1 Test results

Test	Sample qty	Avg	Max	Min
Latency from ISR to waken-up thread	6731	7.5 µs	17.7 µs	7.2 µs

4.6.2.2 Diagrams



4.6.3 Maximum sustained interrupt frequency (IRQ_S_SUS)

The “maximum sustained interrupts frequency” test measures the probability that an interrupt might be missed. It attempts to answer the question: Is the interrupt handling duration stable and predictable?

This test is done in 3 phases:

- 1000 interrupts as an initial phase: a fast test just to see where we have to start searching.
- 1 000 000 interrupts as a second phase based on the results from the first phase. This test still takes less than a minute and gives already accurate results.
- 1billion interrupts as a last phase, which takes few hours and sometimes more than 24 hours, depending on the used platform and OS. This phase is done to verify stability; therefore, we cannot run this phase many times, especially when it comes to large interrupt latencies.

As one can observe in the previous test results, although the interrupt latency is in the best case 7 µs, the clock tick gives us a penalty. On the long run, you can see that the guaranteed interrupt latency comes around 13µs.

☺ The fact that the latency is very stable is of course good. However, this is also partially caused by the large L2 so that the clock interrupt is never started from main RAM.

4.6.3.1 Test results

Interrupt period	#interrupts generated	#interrupts serviced	#interrupts lost
9 µs	1 000	998	2
11 µs	1 000	999	1
12 µs	1 000	1 000	0
11 µs	1 000 000	999 999	1
12 µs	1 000 000	1 000 000	0
12 µs	1 000 000 000	999 999 995	5
13 µs	1 000 000 000	1 000 000 000	0

4.7 Memory tests

This examines the memory leaks of OS.

4.7.1 Memory leak test (MEM_B_LEK)

This test continuously create/remove objects in the operating system (threads, ***semaphores***, ***mutexes*** …).

Test	result
Test succeeded	YES
Test duration (how long we let the endless loop run)	>10h
Number of main test loops done	> 50 000

5 Appendix A: Vendor comments

All vendor comments were integrated within the document as there were no disagreements.

6 Appendix B: Acronyms

Acronym	Explanation
API	Application Programmers Interface: calls used to call code from a library or system.
BSP	Board Support Package: all code and device drivers to get the OS running on a certain board
DSP	Digital Signal Processor
FIFO	First In First Out: a queuing rule
GPOS	General Purpose Operating System
GUI	Graphical User Interface
IDE	Integrated Development Environment (GUI tool used to develop and debug applications)
IRQ	Interrupt Request
ISR	Interrupt Servicing Routine
MMU	Memory Management Unit
OS	Operating System
PCI	Peripheral Component Interconnect: bus to connect devices, used in all PCs!
PIC	Programmable Interrupt Controller
PMC	PCI Mezzanine Card
PrPMC	Processor PMC: a PMC with the processor
RTOS	Real-Time Operating System
SDK	Software Development Kit
SoC	System on a Chip