LINUX 2.6.33.7.2-RT30 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

Disclaimer

Although all care has been taken to obtain correct information and accurate test results, Dedicated Systems Experts, VUB-Brussels, RMA-Brussels and the authors cannot be liable for any incidental or consequential damages (including damages for loss of business, profits or the like) arising out of the use of the information provided in this report, even if these organisations and authors have been advised of the possibility of such damages.

http://www.dedicated-systems.com

E-mail: info@dedicated-systems.com

EVALUATION REPORT LICENSE

This is a legal agreement between you (the downloader of this document) and/or your company and the company DEDICATED SYSTEMS EXPERTS NV, Diepenbeemd 5, B-1650 Beersel, Belgium.
It is not possible to download this document without registering and accepting this agreement on-line.

1. **GRANT**. Subject to the provisions contained herein, Dedicated Systems Experts hereby grants you a non-exclusive license to use its accompanying proprietary evaluation report for projects where you or your company are involved as major contractor or subcontractor. You are not entitled to support or telephone assistance in connection with this license.
2. **PRODUCT**. Dedicated Systems Experts shall furnish the evaluation report to you electronically via Internet. This license does not grant you any right to any enhancement or update to the document.
3. **TITLE**. Title, ownership rights, and intellectual property rights in and to the document shall remain in Dedicated Systems Experts and/or its suppliers or evaluated product manufacturers. The copyright laws of Belgium and all international copyright treaties protect the documents.
4. **CONTENT**. Title, ownership rights, and an intellectual property right in and to the content accessed through the document is the property of the applicable content owner and may be protected by applicable copyright or other law. This License gives you no rights to such content.
5. **YOU CANNOT**:
   - You cannot, make (or allow anyone else make) copies, whether digital, printed, photographic or others, except for backup reasons. The number of copies should be limited to 2. The copies should be exact replicates of the original (in paper or electronic format) with all copyright notices and logos.
   - You cannot, place (or allow anyone else place) the evaluation report on an electronic board or other form of on line service without authorisation.
6. **INDEMNIFICATION**. You agree to indemnify and hold harmless Dedicated Systems Experts against any damages or liability of any kind arising from any use of this product other than the permitted uses specified in this agreement.
7. **DISCLAIMER OF WARRANTY**. All documents published by Dedicated Systems Experts on the World Wide Web Server or by any other means are provided "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. This disclaimer of warranty constitutes an essential part of the agreement.
8. **LIMITATION OF LIABILITY**. Neither Dedicated Systems Experts nor any of its directors, employees, partners or agents shall, under any circumstances, be liable to any person for any special, incidental, indirect or consequential damages, including, without limitation, damages resulting from use of OR RELIANCE ON the INFORMATION presented, loss of profits or revenues or costs of replacement goods, even if informed in advance of the possibility of such damages.
9. **ACCURACY OF INFORMATION**. Every effort has been made to ensure the accuracy of the information presented herein. However Dedicated Systems Experts assumes no responsibility for the accuracy of the information. Product information is subject to change without notice. Changes, if any, will be incorporated in new editions of these publications. Dedicated Systems Experts may make improvements and/or changes in the products and/or the programs described in these publications at any time without notice. Mention of non-Dedicated Systems Experts products or services is for information purposes only and constitutes neither an endorsement nor a recommendation.
10. **JURISDICTION**. In case of any problems, the court of BRUSSELS-BELGIUM will have exclusive jurisdiction.

Agreed by downloading the document via the internet.

DOCUMENT CHANGE LOG

Issue No.	Revised Issue Date	Para's / Pages Affected	Reason for Change
1.0	25 Mar 2011	All	Initial version
1.01	2 April 2011	some	Typo changes
1.02	14 April 2011	ALL	Rephrasing some statements and commenting
1.03	03 May 2011	ALL	idem
1.04	05 May 2011	ALL	Idem
1.05	13 May 2011	ALL	Almost-Final
1.06	16 May 2011	ALL	Almost-Final + just few changes more
1.07	30 May 2011	All	Final

1 Document Intention

1.1 Purpose and scope

This document presents the quantitative evaluation results of the real-time **Linux** operating system (**Linux** with its real-time patches). The testing results of this operating system employed on an x86 processors can be found on our website. (www.dedicated-systems.com)

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]. See section 0 of this document for more detailed references. 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 0 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 Document issue: the 2.9 framework

This document shows the test results in the scope of the evaluation framework 2.9.

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

Those are the 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 Linux 2.6.33.7.2-RT30
This document presents the qualitative discussion of the OS
1EVA-2.9-OS-LNX-104 Issue: 1 May 13, 2011

2 Introduction

This chapter talks about: 1) the OS that we are going to test and evaluate, 2) the real time patch integrated in this OS to achieve some real time performance and behaviour tests, 3) the library used for interaction between the testing applications and the kernel, 4) the hardware on which the under testing OS will be employed.

2.1 Overview

The evaluation project started in 1995 and as such accumulates a long experience with different (RT) OS. Today more and more embedded systems are equipped with **Linux** solutions using more or less real-time patches. Different vendors like MontaVista, Windriver, and Lynuxworks have now **Linux** variants in their product portfolio.

Since the kernel version 2.4, a lot of improvements regarding real-time behaviour found their way into the standard "**Vanilla**" kernel. There is a well maintained real-time patch available (both have their origins from Ingo Molnar) called ***RT_PREEMPT*** patch. Remark that some real-time features (like priority-inheritance ***mutexes***, introduced in version 2.6.18) are already in the **Vanilla** kernel.

We believed that it is the time to test this kernel by our standard real-time behaviour evaluation framework and find out how well it behaves.

For this evaluation, we used the standard **glibc** library as the ***µ*Clibc** package does not include yet the **Native POSIX Thread Library (NPTL**). Moreover, **µClibc** also does not use **futexes** which means that it also does not have any support for priority inheritance (which must be available when considering real-time behaviour). Further **µClibc** uses internal protection systems (***mutex***, ***semaphores***) and signals for the ***pthread POSIX*** layer, which behaves differently while compared to the usage of direct `NPTL calls`. From a real-time point of view, using **µClibc** in its current form makes the kernel real-time support unavailable in user space. It has to be said that there is an active **NPTL** branch in the **µClibc** code base. Therefore we suspect that it is only a matter of time before it will become available in the official releases.

It is a pity that the **µClibc** wagon is not yet on the **NPTL** rail. Using the **buildroot**, **µClibc**, and **busybox** combo makes it easier to have an embedded **Linux** platform with a small storage footprint. But without the **NPTL** support, real-time applications cannot be used in user space, unless you use direct system calls to the kernel.

Remark that the ***RT_PREEMPT*** patch degrades throughput performance which means that it should be used only when your project has low latency requirements. This is normal and a fundamental rule in real-time software: latency improvements have a negative impact on throughput and vice versa. Some quick measurements using an NFS mount stressing network and disk showed a negative throughput impact between 5 and 10% by enabling the ***RT_PREEMPT*** patch!

2.2 Evaluated (RTOS) product

2.2.1 Software

The operating system OS that will be evaluated is **Vanilla Linux** 2.6.33.7 with real-time patch v30.This RT patch was the latest version officially released by OSADL (the Open Source Automation Development Lab) on December 21, 2010. Being as OSDAL's latest stable release was our main reason for testing this version. The RT patches can be found at http://www.kernel.org/pub/linux/kernel/projects/rt/.

The evaluation of this kernel version (2.6.33.7.2-rt30) was performed using several performance and behaviour tests. The testing results are applicable only to this version as other versions may have other significant performance figures and behaviour.

The library used between the testing applications and the kernel is the **glibc** version 2.11.1 as mentioned before. This interfacing library is important because user applications (when using POSIX calls) can access the real-time features of the kernel only if this library supports them. Otherwise, direct system calls in user space applications are needed.

2.2.2 Hardware

The operating system was tested on our standard x86 Pentium MMX evaluation platform running at 200MHz with 128MB RAM. So we can compare the results with other RTOS tested on the same platform.

This platform is already pretty old (from 1997 to be correct), but its advantage is that it makes tests comparable as we have tested all RTOS for more than a decade on this platform. Also there is no impact from BIOS interrupts (motherboard/BIOS is too old).

3 Evaluation results summary

Remember that the tested and evaluated product is **Vanilla Linux** 26.33.7 with ***RT_PREEMPT*** patch v30. If correctly used and configured, the ***RT_PREEMPT*** **Linux** system has the internals to provide some real-time characteristics.

Compared with the traditional RTOS that supports also memory protection between processes, the worst case latencies in **Linux** ***RT_PREEMPT*** are still around 5 to 10 times slower (depending on the RTOS you compare with). Our study and measurements show the latencies are bound and therefore this **Linux** version may be labelled Real-Time.

TAKE CARE: Using a wrong driver or wrong configuration can destroy real-time behaviour. You need to follow the detailed rules described in the relevant document (Doc 5).

3.1 Positive points

- No license fees
- Source code available
- Extensible

3.2 Negative points

- The real-time characteristics of the OS are present only when everything is configured and built correctly (and not for all drivers)
- GPL license is not completely free…
- Setting up a complete embedded target from scratch is a daunting task.
- **uClibc**, which is used a lot in embedded systems, does not have currently NTPL support and as such cannot provide real-time characteristics to the user level. Thus, glibc should be used.

3.3 Ratings

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

RTOS Architecture	0	6	10
OS Documentation	0	4	10
OS Configuration	0	6	10
Internet Components	0	10	10
Development Tools	0	6	10
Installation and BSP	0	4	10
Test Results	0	4	10
Support	0	N.A.	10

Although [Doc. 3] gives a description of the ratings, comparison with other reports on other OS should help you understand the scoring.

4 Test Results

Test Results 0 4 10

Compared with traditional RTOS supporting inter-process memory as well, the worst case latencies in **Linux *RT_PREEMPT*** *are around 5 to 10 times greater (depending on the RTOS you compare with). Still they are inbound and thus considered as real-time!*

However, using wrong configuration (at build and/or at runtime) or incorrect behaving drivers can still destroy the potential real-time behaviour.

One of the items that are surely candidate to be improved is the clock tick interrupt duration, which is the Achilles' heel of **Linux *RT_PREEMPT***.

4.1 Calibration system test (CAL)

These tests are used to calibrate the tracing overhead compared with the processing power of the platform. This is important to understand the accuracy of the measurements done in scope of this report.

They are also used 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)

This test calibrates the tracing system overhead. This is more hardware than OS related, but it is needed to correct the measured times.

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

Tracing accuracy depends here on the PCI clock (33MHz), as this is the minimum time frame that can be detected. In general, the results in this document 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	209 nsec
minimum tracing overhead	209 nsec
maximum tracing overhead	209 nsec

4.1.2 CPU power (CAL-P-CPU)

This test will calibrate the CPU performance and the memory bandwidth of the used platform. This test is measured in different situations, 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 172KB 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 a “feeling” about the impact of page faults, even if the test software is launched from a RAM file system and uses `mlockall`.

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).

Due to the rework, the impact of being/ or not being in the I-Cache has enlarged enormously compared with previous tests.

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

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

Further, we can adapt the test for CPU which has larger cache sizes as the arrays have to be larger than the cache size (across all levels).

4.1.2.1 Test results

The results for our standard platform (Pentium MMX 200 MHz) are shown below:

Test	no cache	cached	cache effect
CPU test: first load, page faults	1562 us	272.7 us	
CPU test: ICache effect	1504 us	271.7 us	5.5
MEM write test	815.2 us	738.6 us	1.1
MEM read test	1355 us	817.1 us	1.7
Average caching effect (CPU and MEM)			2.8

Here are some conclusions regarding the Pentium MMX 200MHz:

- 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.
- Compared with the situation of the instructions absence in the cache, page faults are a minor issue (in this case around 5%). Remark that we are running from a RAM file system and that the page fault handler will be in the cache anyhow during this load.
- 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 a huge 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.

4.2 Clock tests (CLK)

The clock test measures the time that an operating system needs 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)

This test is done in order to examine 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.

For this test, the `nanosleep ()` POSIX function call is used. Following POSIX, the delay should be based on the clock tick. The "`nanosleep`" function always pauses for at least its specified time, but however it can take up to one clock tick more than its specified time until the process becomes run-able again".

As the kernel is running at 1000Hz, we expect a clock interrupt each ms. The following table shows the test results.

Test	result
Test succeeded	Yes
Tested clock period	1ms
Clock period adaptable	YES (by kernel config)

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

This test is done for examining 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 tickless kernel will not even prevent this from happening (it will only lower the number of occurrences). The kernel under test was not using the tickless 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.

☹ A clock time duration of about 20 µs is measured, which is bad compared with the traditional RTOS (at least a factor 5 to even 10). This clock will impact all other real-time behaviour and measurements.

This was the main reason for doing this as an initial measurement. Running tests that flush the instruction and data Cache increase this further. We detected durations up to 55us in our tests.

Comparing this with the kernel version 2.4 which had clock tick duration of around 8µs on this platform shows that this should be much better. Clearly, this is the critical path of the current real-time behaviour as all other issues have been improved vastly compared with the older kernels.

4.2.2.1 Test results

Test	result
CLOCK_LOOP_COUNTER	10000
Normal busy loop time	124 µs
Busy loop time with clock interrupt	143 µs, worst case 145 µs
Clock interrupt duration	19 µs to 21µs, detected in some tests up to 55µs impact.

4.2.2.2 Diagrams

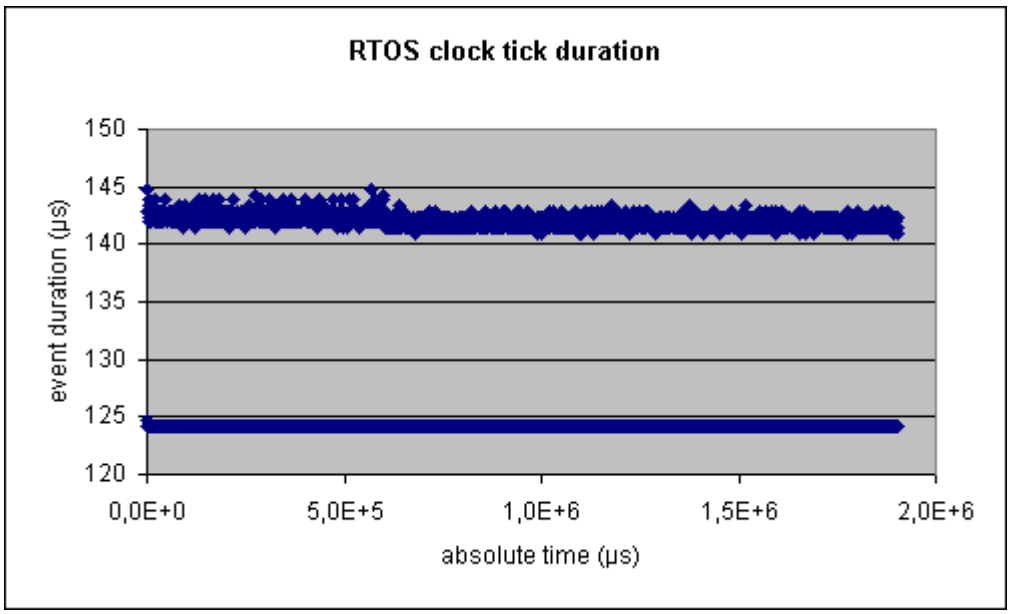


Figure 1: RTOS clock tick duration (1)

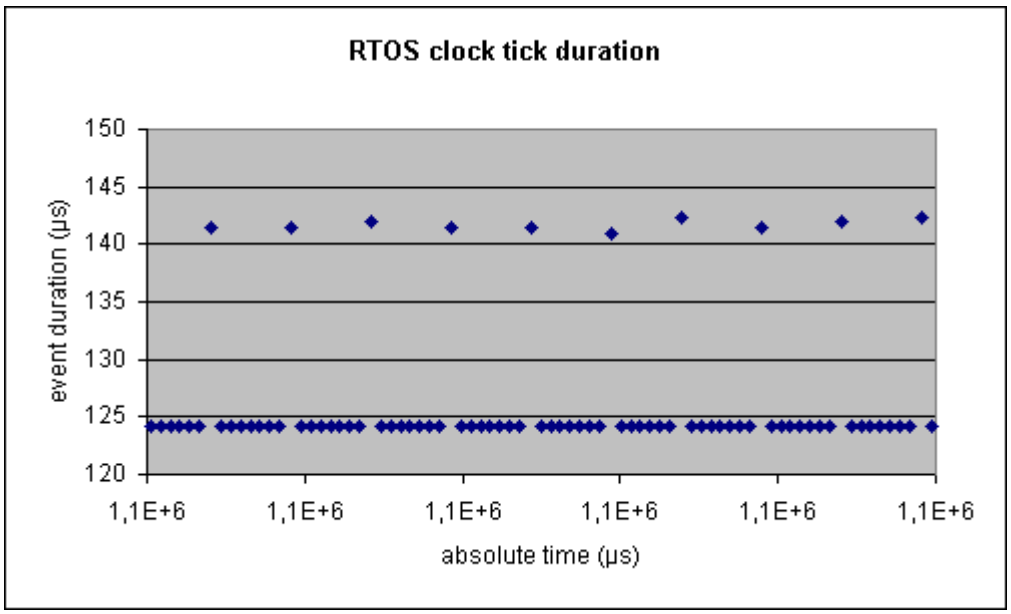


Figure 2: RTOS clock tick duration (2)

4.3 Thread tests (THR)

These tests are used to measure the performance of the scheduler.

☹ Although most of the tests run pretty stable (deterministic), we found some strange issues:

- The queuing behaviour on the ready queue while using ***SCHED_FIFO*** or ***SCHED_RR*** policy is different
- The first ***Round-Robin*** time slice of thread execution takes almost ten times more than the normal time slice.
- The first thread being deleted before it ever run in a process took consistently takes around 5ms, which is about 15 times longer than the other cases.

In real-time design, it is a bad practice to: 1) dynamically create and terminate threads, 2) use multiple real-time threads on the same priority. Therefore, these problems should not popup in a good real-time OS designs but also shouldn't be avoided or ignored by real-time OS designers.

4.3.1 Thread creation behaviour (THR-B-NEW)

This test will examine the behaviour of creating threads. Does the operating system behave as it should be as long as it is considered being a real-time operating system?

This test succeeded, however we noticed again a different behaviour when using the ***SCHED_FIFO*** compared to the ***SCHED_RR*** class! When lowering the priority of a thread:

- It will become in the front of the ready queue if running in ***SCHED_RR*** policy
- It will become at the tail of the ready queue if running in ***SCHED_FIFO*** policy

Not that this is fundamental issue as it is not a good practice in real-time design to use the same priority for different threads. But it is still something to keep in mind.

4.3.1.1 Test results

Test	result
Test succeeded	YES
Lower priority not activated?	YES
Same priority at tail?	This depends if ***SCHED_FIFO*** or ***SCHED_RR*** scheduling class is used for the threads: RR puts it in front of its ready queue FIFO puts it at tail of its ready queue
Yielding works?	YES
Higher priority activated?	YES

4.3.2 Round robin behaviour (THR-B-RR)

This test checks if the scheduler uses a fair ***Round Robin*** mechanism when threads are having the same priority and all of them are in the ready-to-run state (and using the ***SCHED_RR*** scheduling policy)!

☹ A problem was discovered here with the time slicing mechanism. When a thread is created (and thus put at the front of the ready queue) it seems to run for around 1s before giving the CPU back to the next thread in the ready queue with same priority.

Once all the threads are running, the time slice goes back to a 10Hz frequency.

This can be clearly seen in the trace file when this test is done with 10 threads:

```
Sample     TimeAbs    TimeRel Data
  TRIG:       0.0ns      0.0ns F0000009
     1:   983.67ms   983.67ms F0000008
     2:     1.967s   983.67ms F0000007
     3:     2.951s   983.67ms F0000006
     4:     3.935s   983.67ms F0000005
     5:     4.918s   983.67ms F0000004
     6:     5.902s   983.67ms F0000003
     7:     6.886s   983.67ms F0000002
     8:     7.869s   983.67ms F0000001
     9:     8.853s   983.67ms F0000000
    10:     9.837s   983.67ms F0000009
    11:     9.936s   99.438ms F0000008
    12:     10.04s   99.438ms F0000007
    13:     10.14s   99.438ms F0000006
    14:     10.23s   99.438ms F0000005
```

Trace extract showing strange round robin behaviour.

4.3.2.1 Test results

Test	result
Test succeeded	NO: first yield upon thread creation uses 983ms and next yields take 99ms!
RR Time slice following this test	100 ms

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

This test measures the time to switch between threads of the same priority. Therefore, threads have to yield the processor voluntary for the other threads for using it.

In this test, we use the ***SCHED_FIFO*** policy; otherwise it would be possible that a ***Round-Robin*** clock event occurs between the yield and the trace, so that the thread activation is not seen in the trace.

This test was performed several times, and each time using a higher number of threads in order to generate the worst case behaviour. If more threads are active, the caching effect will be obvious in a way that the thread context will not reside anymore in the cache once we have enough threads (on this platform the caches are only 16K, both for the data as the instruction cache).

Further you will see clearly the influence of clock interrupts (causing the maximum values in the graphics). As loading/starting the test software passes a lot of code and data in the kernel, the next clock interrupt will not be cached (causing the 50µs delay for the first clock tick in the 2/10 thread scenario). Once there are enough running threads, the clock interrupt will always be un-cached (at least data part of it) and thus for the 128/1000 thread tests, the clock interrupts always generate a delay of approximately 50µs.

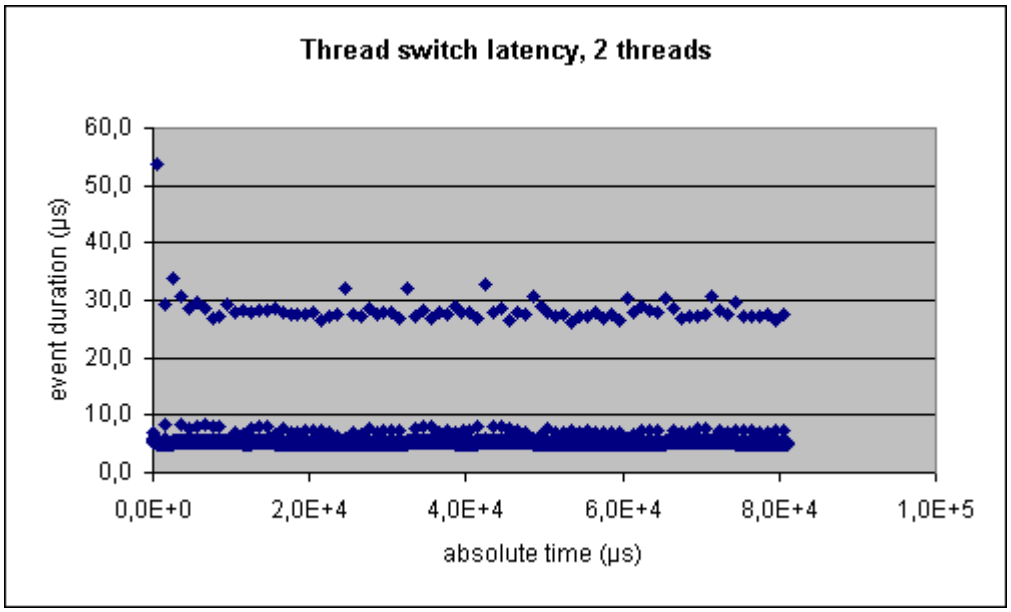
For the rest, the thread switch latency is a fairly stable line, which is good.

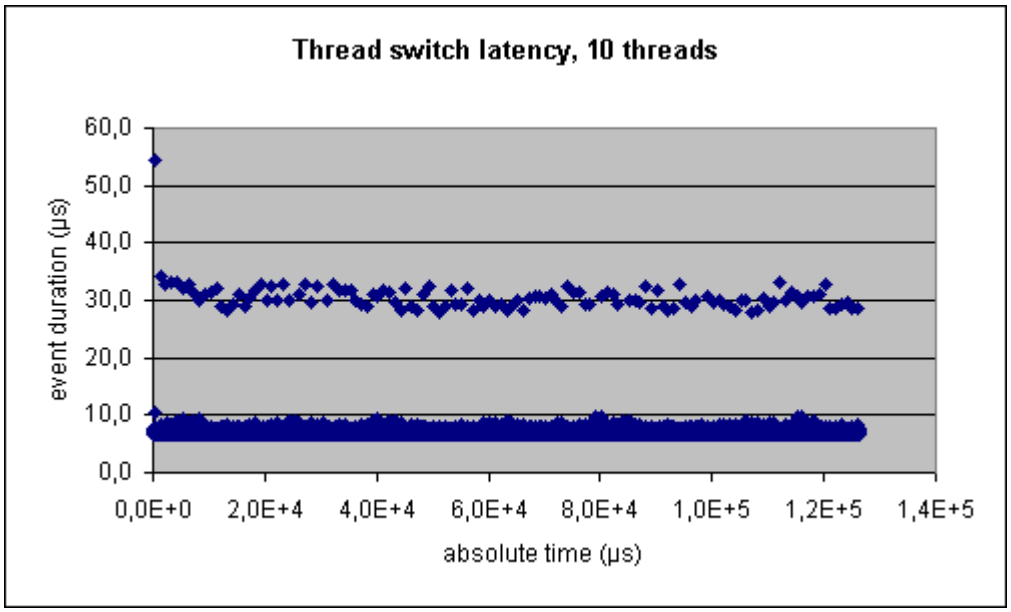
4.3.3.1 Test results

Test	result
Test succeeded	YES

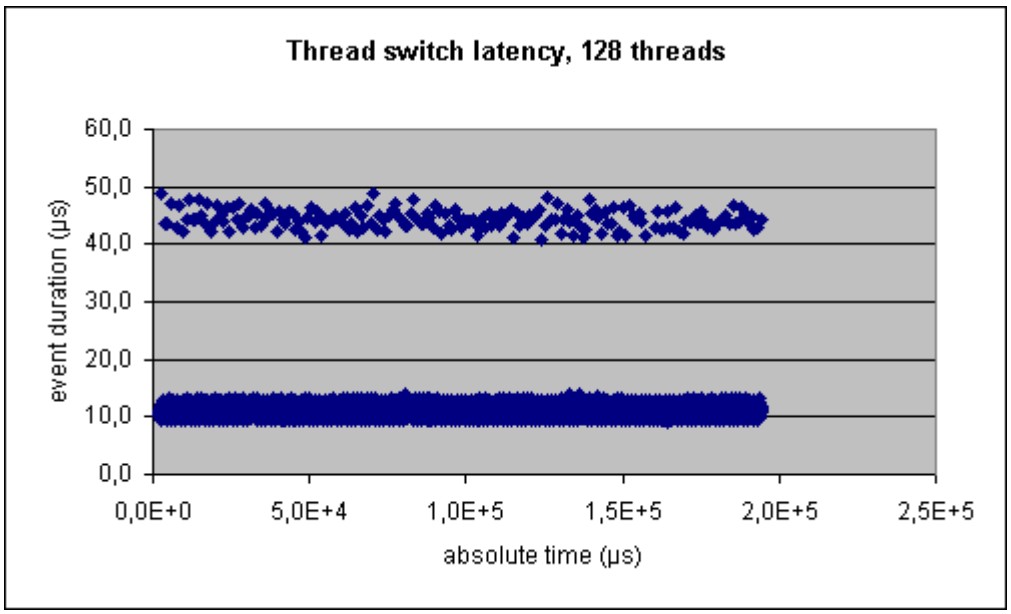
Test	Sample qty	Avg	Max	Min
Thread switch latency, 2 threads	14099	5.3 µs	53.8 µs	4.8 µs
Thread switch latency, 10 threads	16379	7.3 µs	54.4 µs	6.7 µs
Thread switch latency, 128 threads	16320	11.4 µs	49.0 µs	9.2 µs
Thread switch latency, 1000 threads	15917	13.4 µs	53.4 µs	10.6 µs

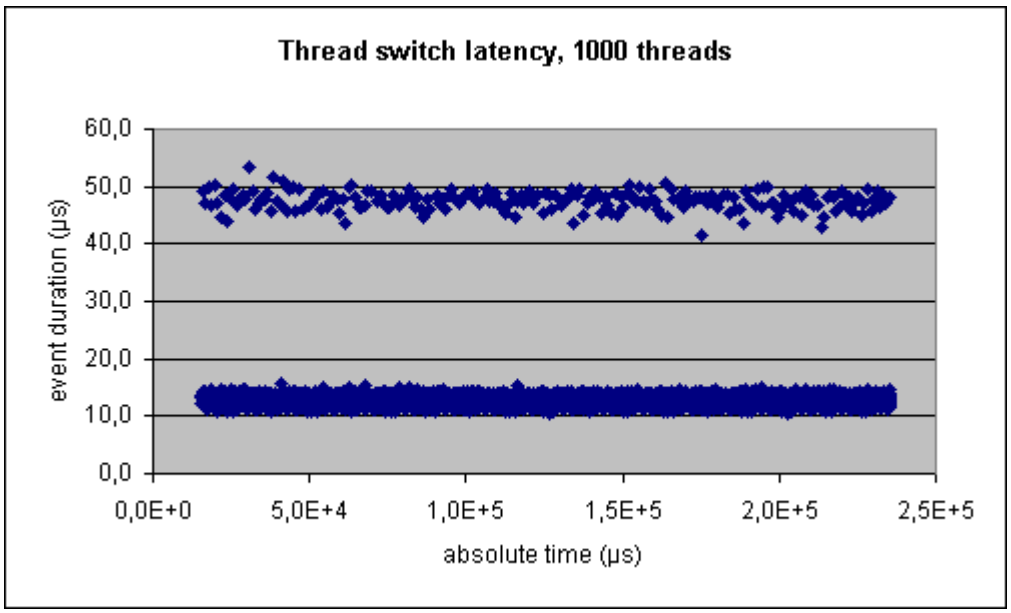
4.3.3.2 Diagrams





http://www.dedicated-systems.com
Email: info@dedicated-systems.com





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

This test examines the time for creating a thread, and the time for deleting a thread in 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, 3), the creation time is the duration from the system call creating the thread to the time when the created thread is activated. For the “never run” scenario, the creation time is the duration of the system call.

Here we found a case that consistently took around 5 ms to handle: in the scenario of creating a thread and deleting it again before it ever run (scenario 1), the first thread requires always 5 ms to be deleted.

We adapted the test aiming to figure out how we could avoid this delay, but all the tests always produce this situation. This means that there is a specific behaviour in the kernel (or libraries) that is causing this corner case.

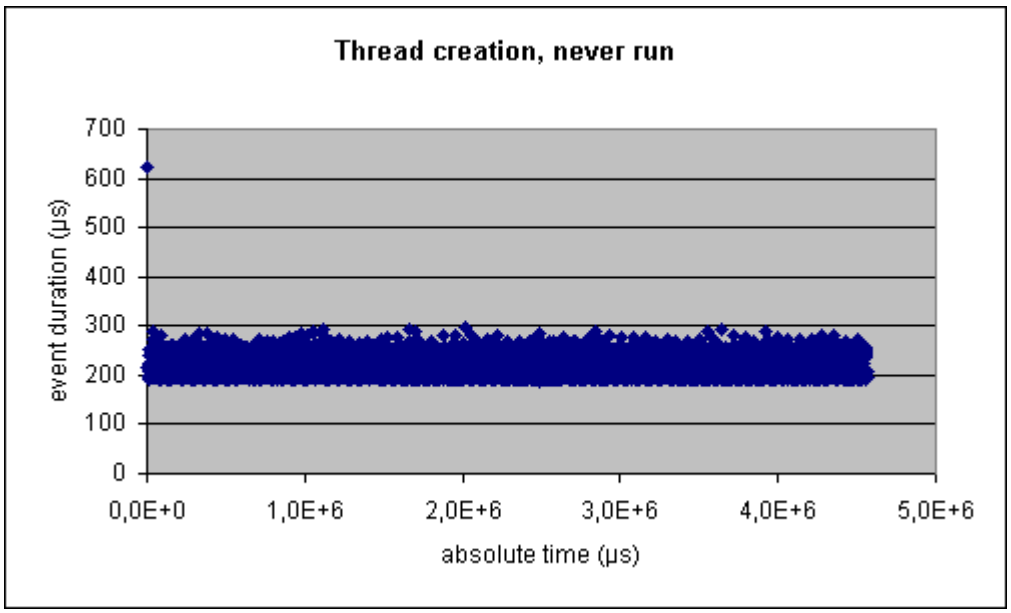
4.3.4.1 Test results

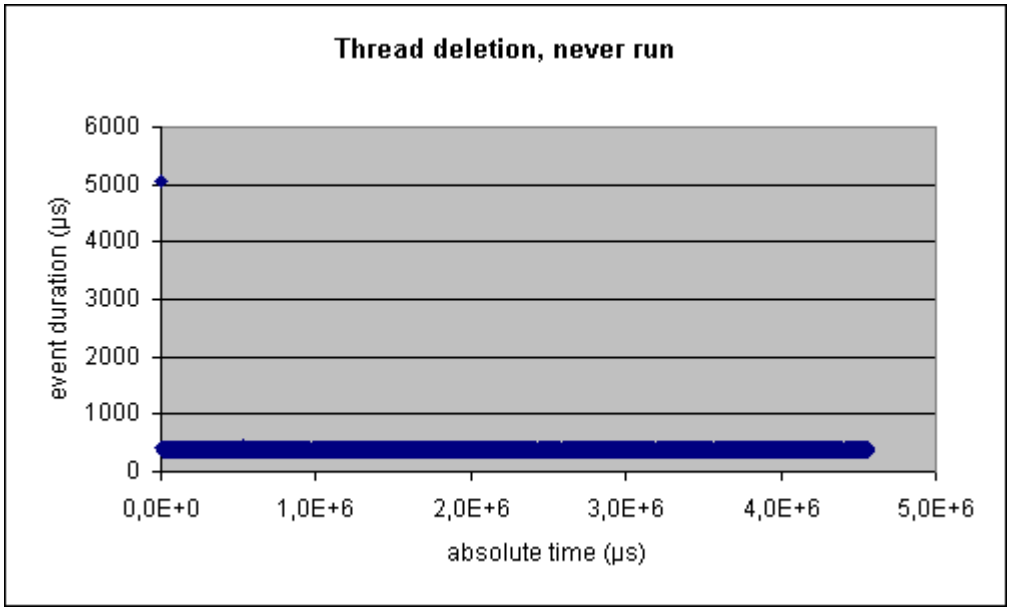
Test	result
Test succeeded	YES

Test	Sample qty	Avg	Max	Min
Thread creation, never run	7500	213.4 µs	621.µs	188.3 µs
Thread deletion, never run	7500	371.7 µs	5044 µs	342.4 µs
Thread creation, run and terminate	7500	339.7 µs	738.5 µs	304.1 µs
Thread deletion, run and terminate	7500	13.7 µs	60.1 µs	11.7 µs
Thread creation, run and block	7500	347.1 µs	736.6 µs	312.7 µs
Thread deletion, run and block	7500	211.8 µs	268.7 µs	89.0 µs

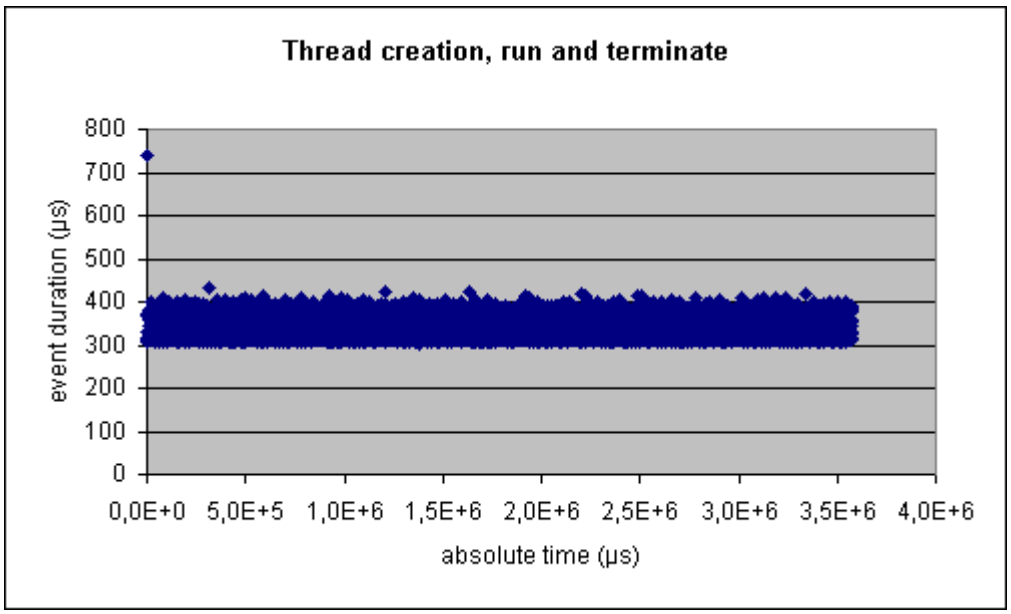
http://www.dedicated-systems.com
Email: info@dedicated-systems.com

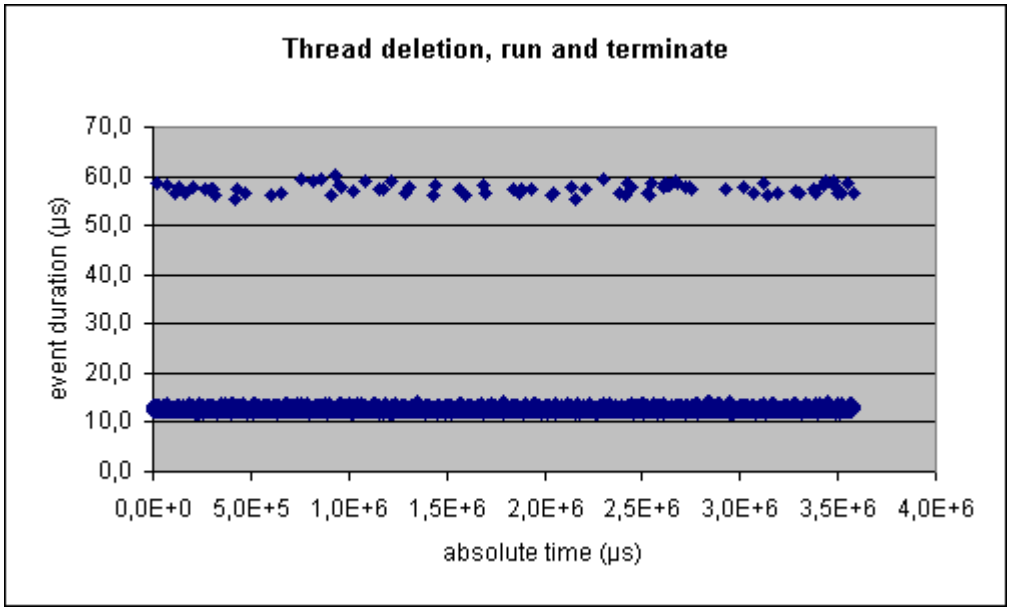
4.3.4.2 Diagrams

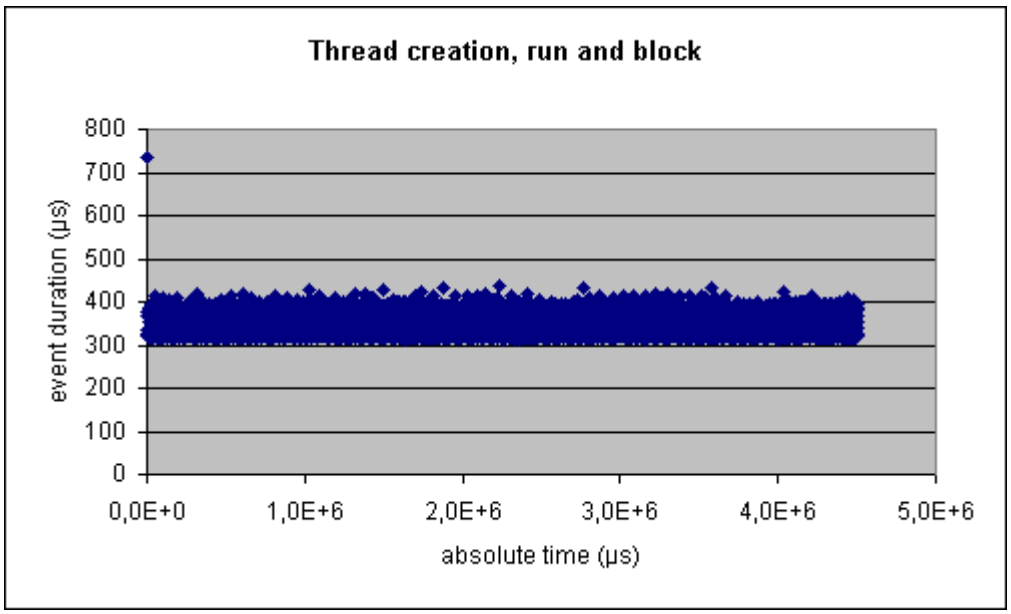


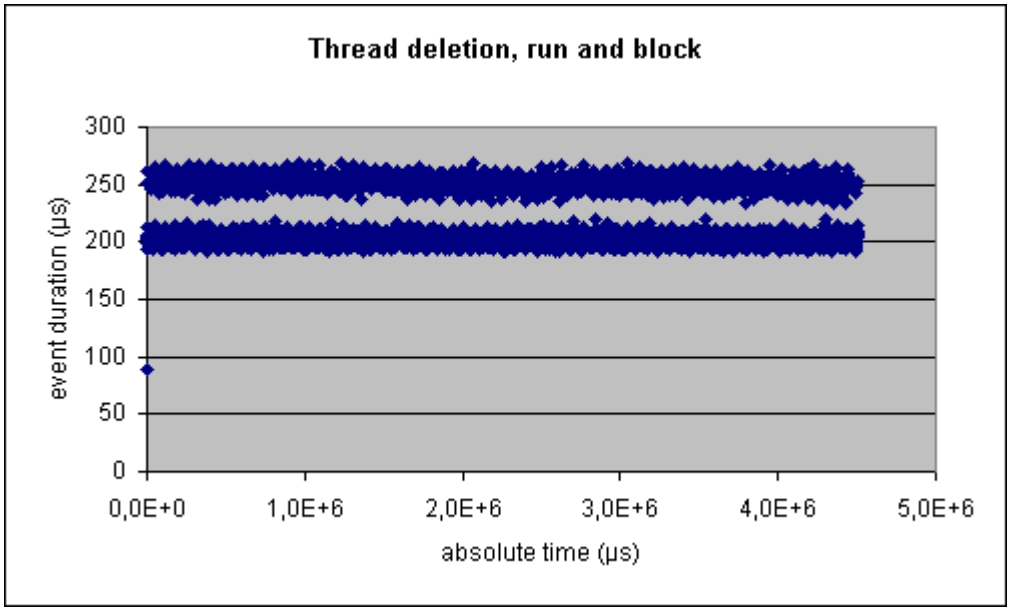


http://www.dedicated-systems.com
Email: info@dedicated-systems.com









4.4 Semaphore tests (SEM)

This test examines the performance and the behaviour of the 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 will experiment if the counting semaphore locking mechanism works as it is expected to do. The `P ()` call should block only when the count is zero. The `V ()` call should increment the semaphore counter. In the case where the semaphore counter is zero, the `V ()` call should cause a rescheduling in the kernel: indeed 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)

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

This **Linux** version 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)

This test is done to get an insight about the time needed to create a ***semaphore*** and the time 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.

For a good RTOS it is expected that there is no difference between the two scenarios. If a difference is detected, then this probably means that the OS handles some initializations on the ***semaphore*** on its first use (making the first use slower).

Anyhow, the tests show that no kernel interaction is needed to create/delete ***semaphores***. The duration of this time is just too small to be measured. The peaks we see are caused by clock interrupts (20/50us).

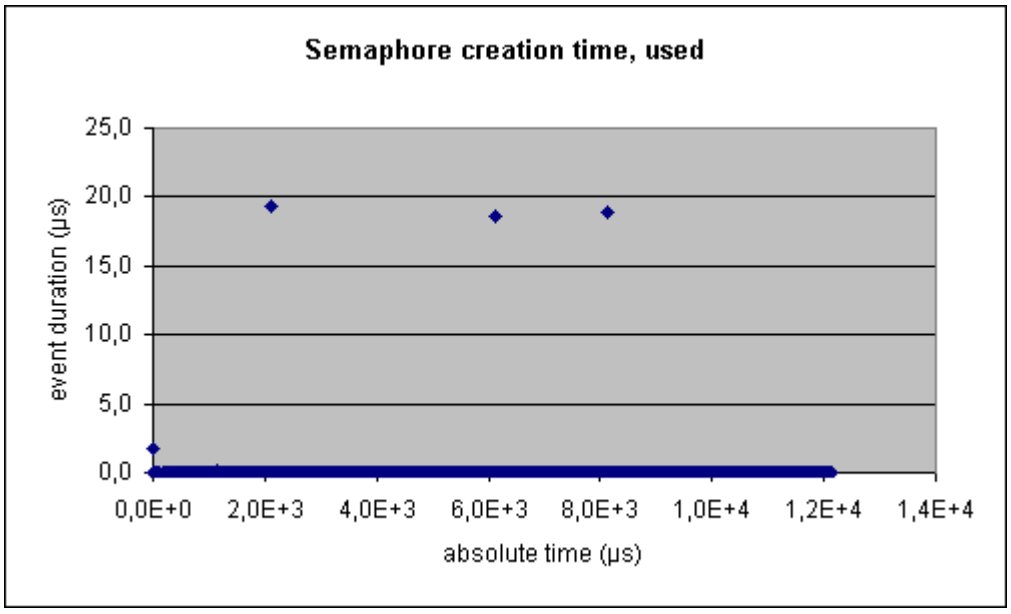
Remark that we do not use “named” ***semaphores***, so they cannot be used between processes. Therefore no access to the kernel is required.

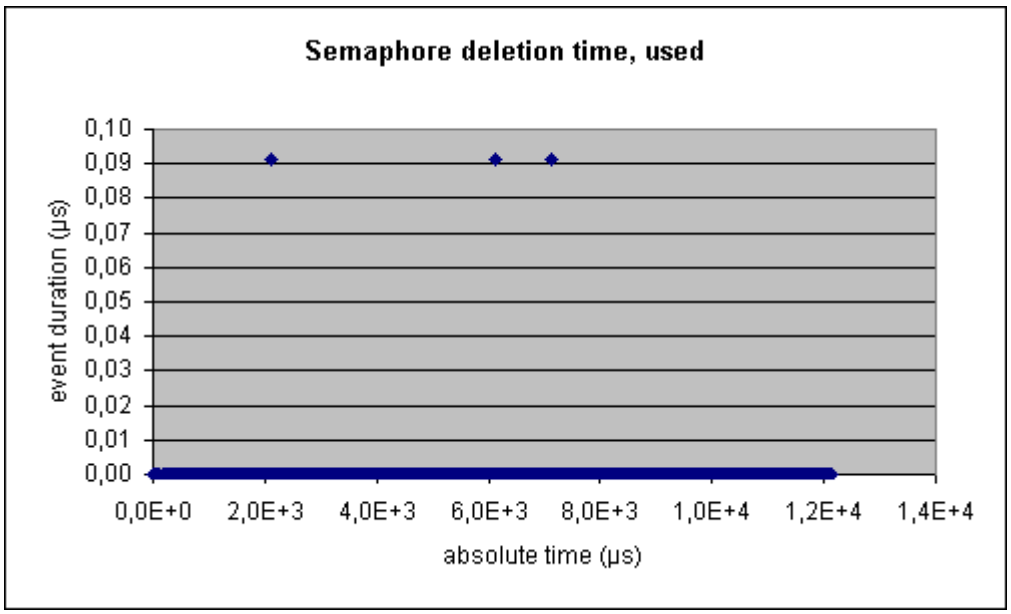
4.4.3.1 Test results

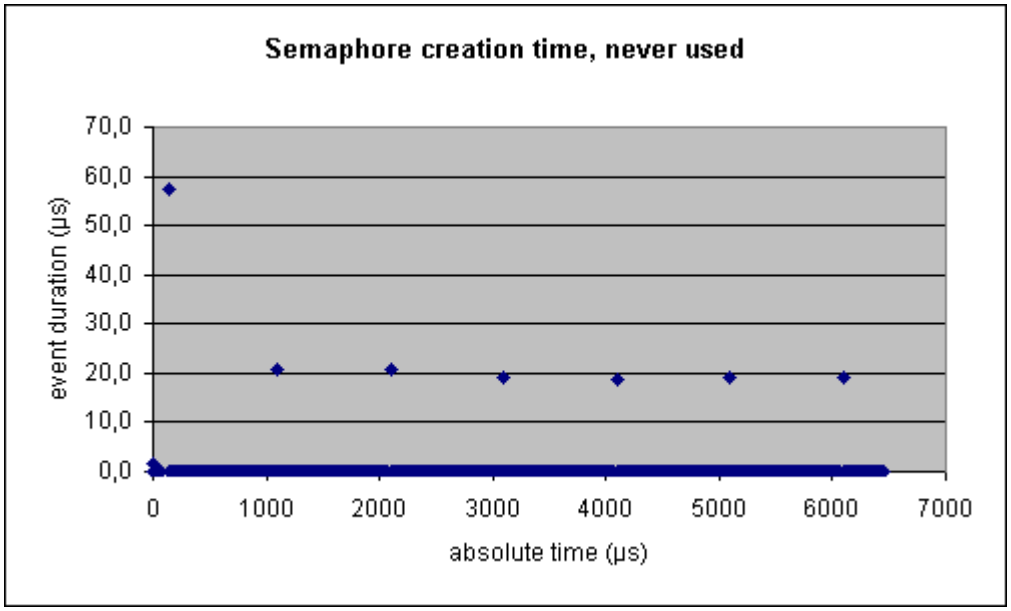
Test	result
Test succeeded	YES

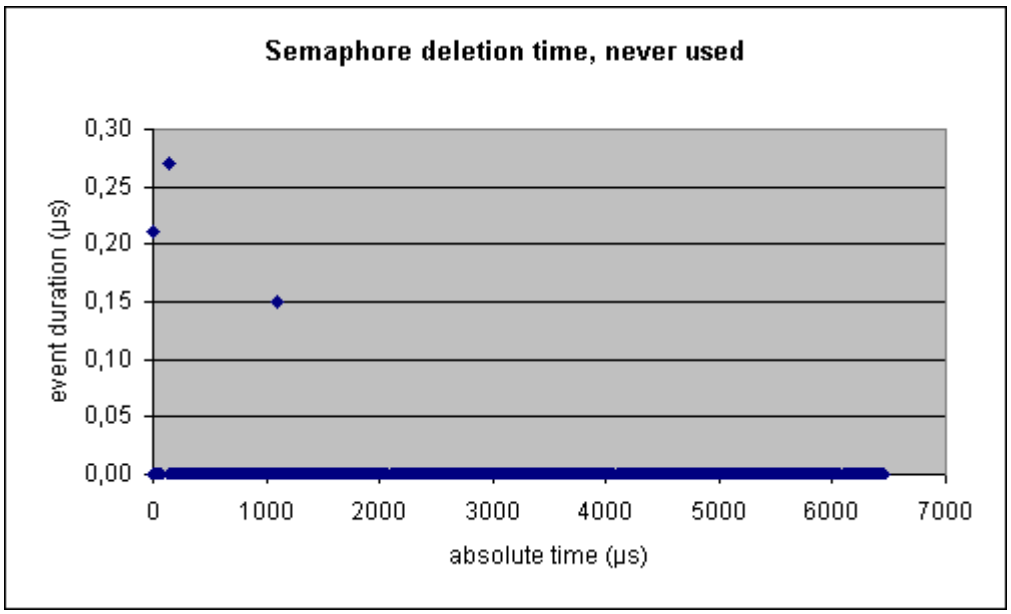
Test	Sample qty	Avg	Max	Min
Semaphore creation time, used	7500	<0.1 µs	19.3 µs	<0.1 µs
Semaphore deletion time, used	7500	<0.1 µs	<0.1 µs	<0.1 µs
Semaphore creation time, never used	7500	<0.1 µs	57.2 µs	<0.1 µs
Semaphore deletion time, never used	7500	<0.1 µs	0.3 µs	<0.1 µs

4.4.3.2 Diagrams









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

Here we test the acquisition and release time in the non-contention case. As in this test case the ***semaphore*** does not neither block nor cause any rescheduling (thread switch), the duration of the call should be very short.

In fact, the library will only need to increase or decrease the ***semaphore*** counter in an atomic way (thus no system call involved). That's why it is too small to be measured.

The clock tick spike is always present

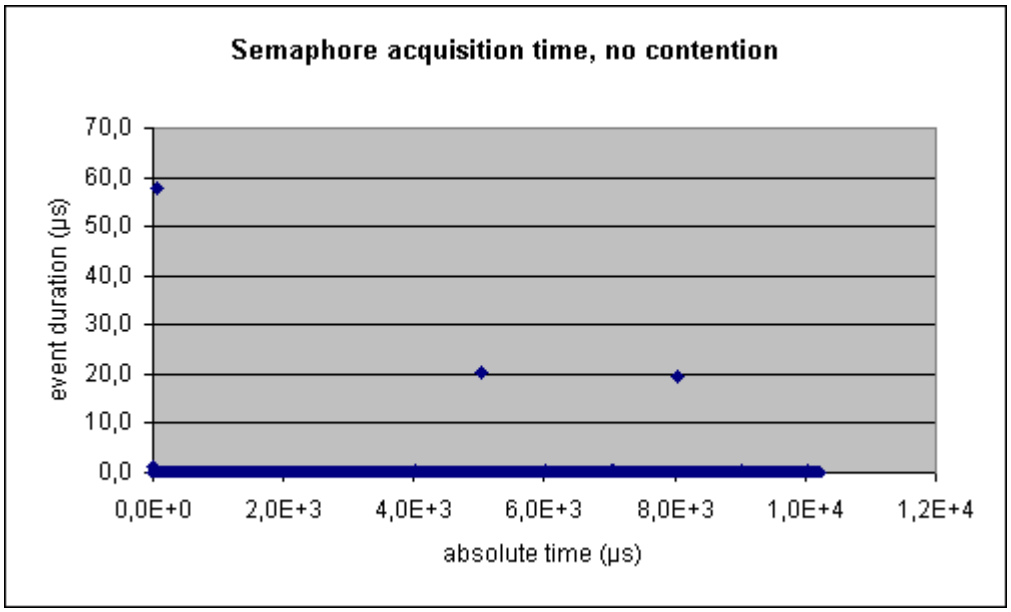
4.4.4.1 Test results

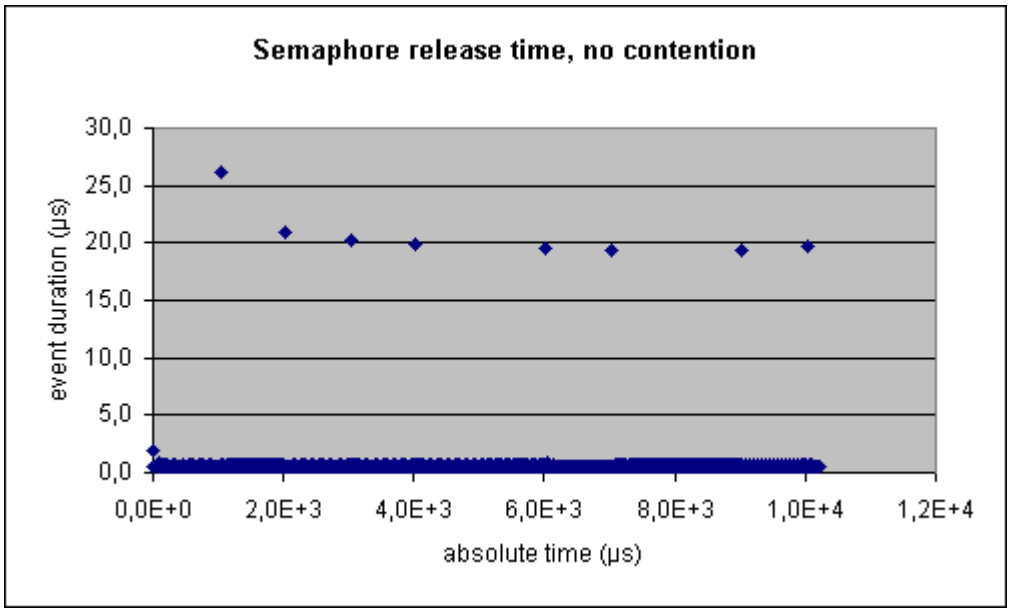
Test	result
Test succeeded	YES

Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, no contention	7500	<0.1 µs	58.0 µs	<0.1 µs
Semaphore release time, no contention	7500	<0.1 µs	26.1 µs	<0.1 µs

http://www.dedicated-systems.com
Email: info@dedicated-systems.com

4.4.4.2 Diagrams





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

This 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 this case happens when the acquisition and release system call causes a rescheduling to occur.

The aim of this test is to verify whether the number of blocked threads has an impact on these timings or not. So this will answer the question: “how much time the operating system needs to find out the next thread to schedule”.

As each thread has a different priority, the question is how these pending thread priorities on a ***semaphore*** are handled. 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 90 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 try (which fails) until 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.
- After the last thread is created and blocked on the ***semaphore***, the creating thread starts to release the ***semaphore***. This happens the same number of times as there are blocked threads.
- 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.

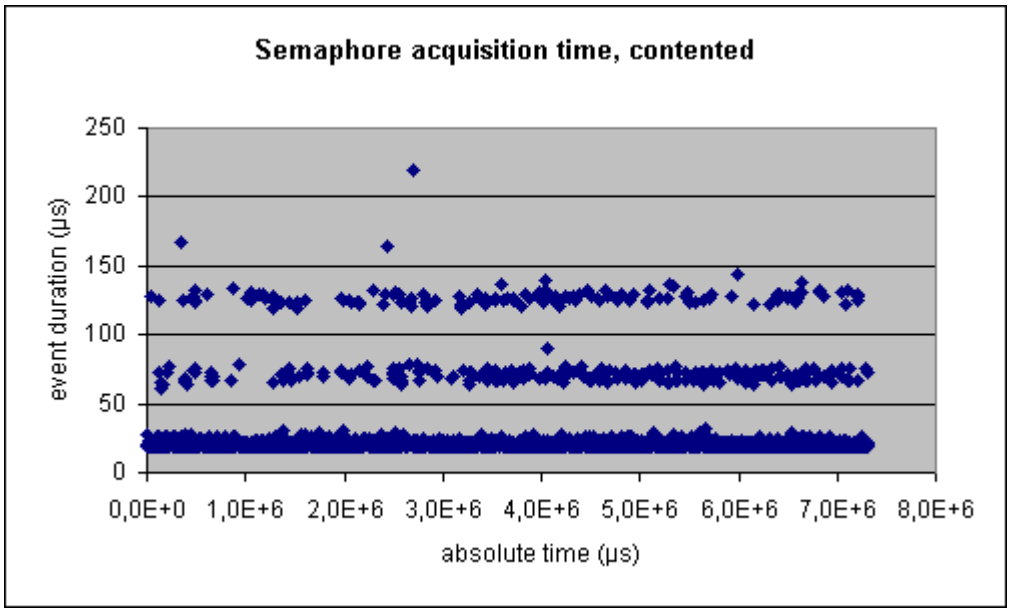
As usual we find the clock tick back in these diagrams, however now we see something strange: it seems that the clock tick can loop a number of times, as we see clearly the distinct lines separated by clock tick duration. This is something strange… and it impacts real-time behaviour.

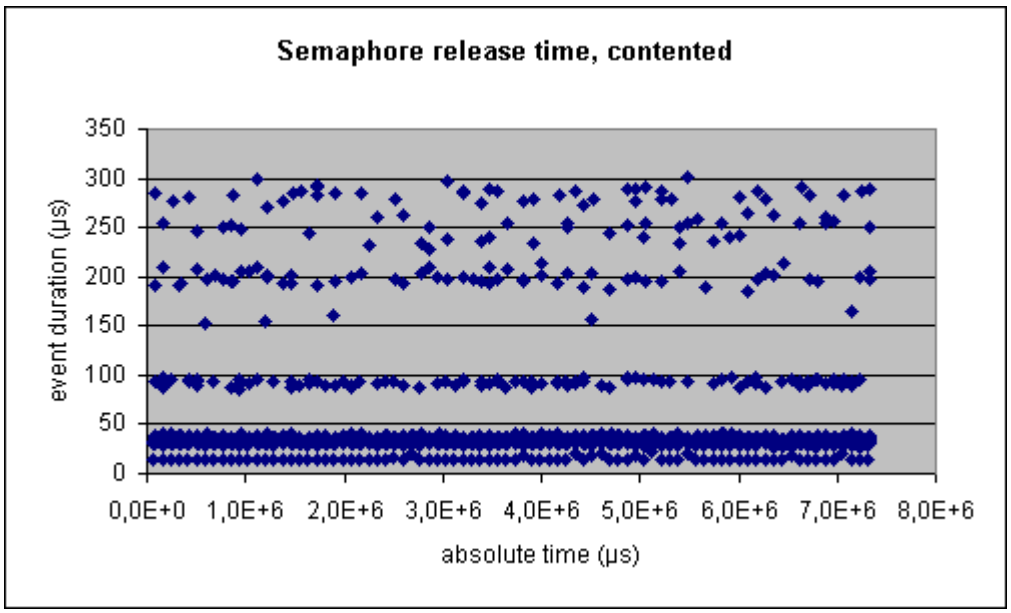
4.4.5.1 Test results

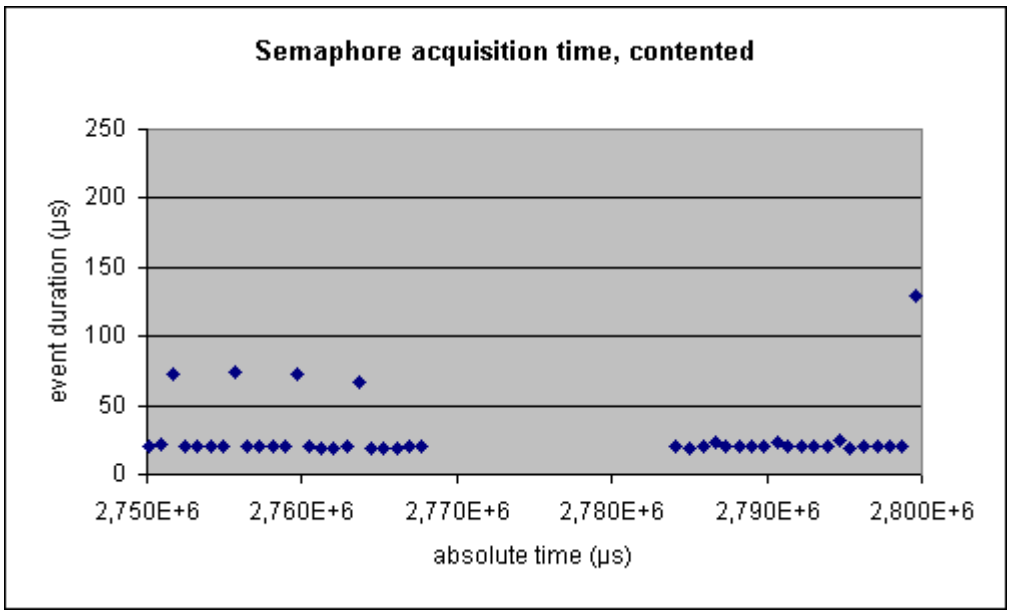
Test	result
Test succeeded	YES
Max number of threads pending	90 (close to the number of priority levels)

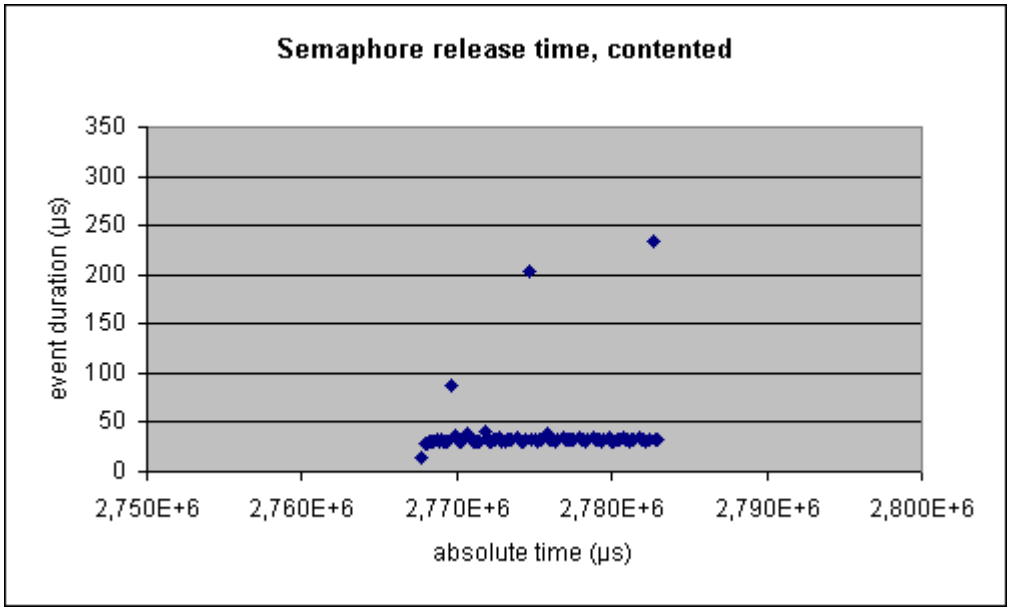
Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, contented	1021	24.01 µs	219.4 µs	18.5 µs
Semaphore release time, contented	1021	36.9 µs	301.2 µs	14.0 µs

4.4.5.2 Diagrams









4.5 Mutex tests (MUT)

Here we are going to test the performance and behaviour of the ***mutual exclusive semaphore***.

Although the mutual exclusive semaphore (further called ***mutex***) is mostly explained as being the same as a counting ***semaphore*** where the count is one, this is not true. A ***mutex*** has a totally different behaviour than ***semaphores***. ***Mutexes*** have the concept of "lock owner", and thus they can be used for preventing priority inversions. This is something that cannot be done by ***semaphores***. Therefore it is a bad idea to use ***semaphores*** as a critical section protection mechanism.

In scope of the framework, this test will look into detail of a ***mutex*** system object that avoids priority inversion.

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

This test will determine if the system call under test prevents the priority inversion case. Therefore the test will artificially create a priority inversion.

As this is one of the first ***RT_PREEMPT*** achievements going into the mainstream kernel, we did not expect any problems. Priority inversion behaves as expected.

4.5.1.1 Test results

Test	result
Priority inversion avoidance system call present	Yes
System call used	pthread_mutex_lock
Test succeeded	YES
Priority inversion avoided	YES
Mechanism used if any?	pthread_mutexattr_setprotocol: PTHREAD_PRIO_INHERIT

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

This is the same test as above, but performed in a loop. In this case, the time is measured to acquire and release the ***mutex*** in the priority inversion case.

Remark that the acquisition enforces a thread switch. The acquiring thread is blocked and the one having the lock is released. The time is measured from the request for the ***mutex*** acquisition to the lower priority thread having the lock being activated.

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).

The release time is measured from the release call to the thread requesting the ***mutex*** being activated.

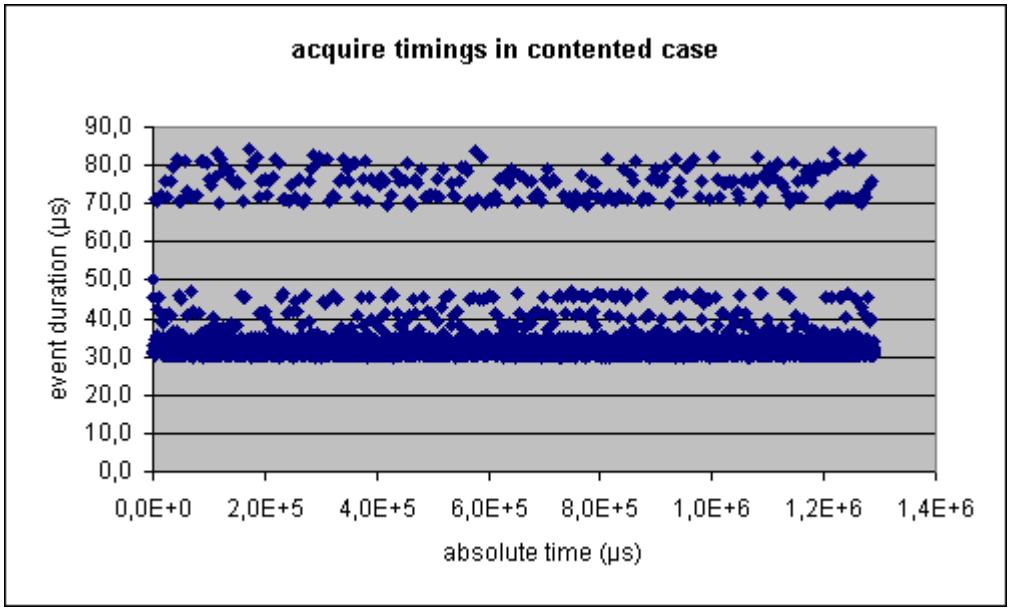
☹ We see that the release seem to cause a double thread switch (time take double as long), which is very strange! None of the other tested RTOS had such behaviour.

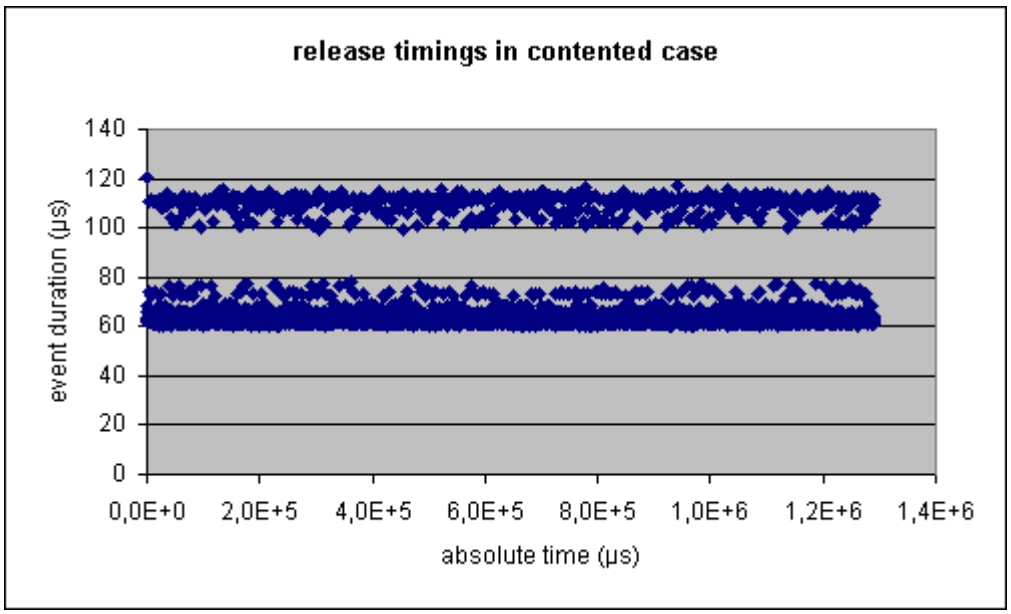
4.5.2.1 Test results

Test	result
Test succeeded	Yes

Test	Sample qty	Avg	Max	Min
Mutex acquisition time, contention	7500	33.2 µs	84.3 µs	29.7 µs
Mutex release time, contention	7500	66.1 µs	120.9 µs	60.1 µs

4.5.2.2 Diagrams:





4.5.3 Mutex acquire-release timings: no-contention case (MUT-P-ARN)

This test measures the overhead of using a lock when it is not locked by another thread. Good 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! We have seen **Linux** running on embedded CPUs that did not have an atomic instruction, which in this case requires a call to the kernel! On such platforms the performance of multi-threaded application, which require locks, is seriously impacted! We observed once an extreme case, where approximately half of the CPU load was caused by the lock-need-to-go-to-the-kernel calls!

As expected, this doesn't take a lot of time, although it takes more time than a ***semaphore***. Maybe the default ***mutex*** attribute is not the FAST one? A ***mutex*** can be configured with different options: recursive, error checking and so one, which can increase the time required to take/release a lock.

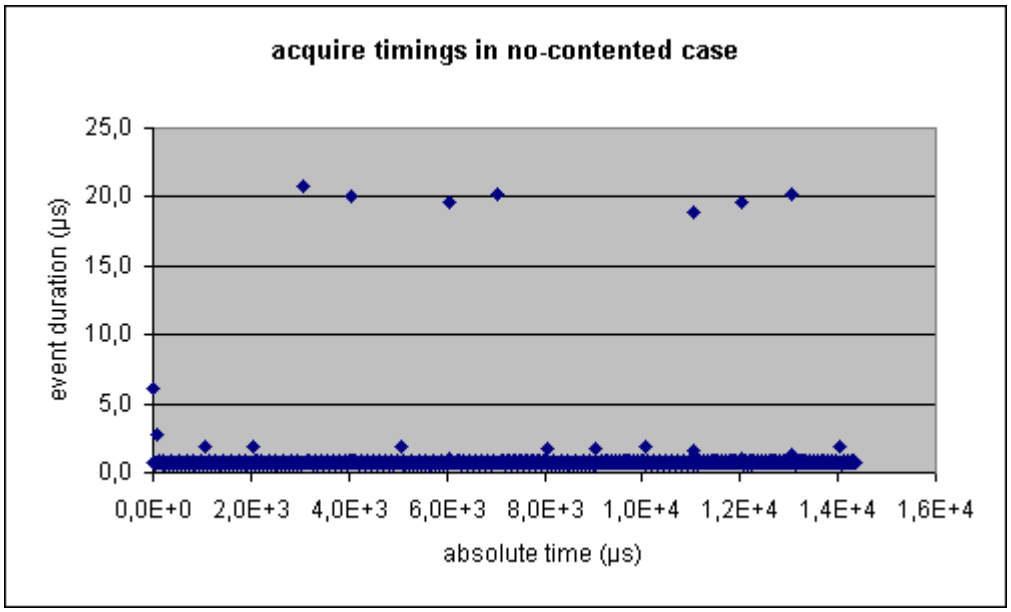
As in all diagrams, the clock tick shows up again.

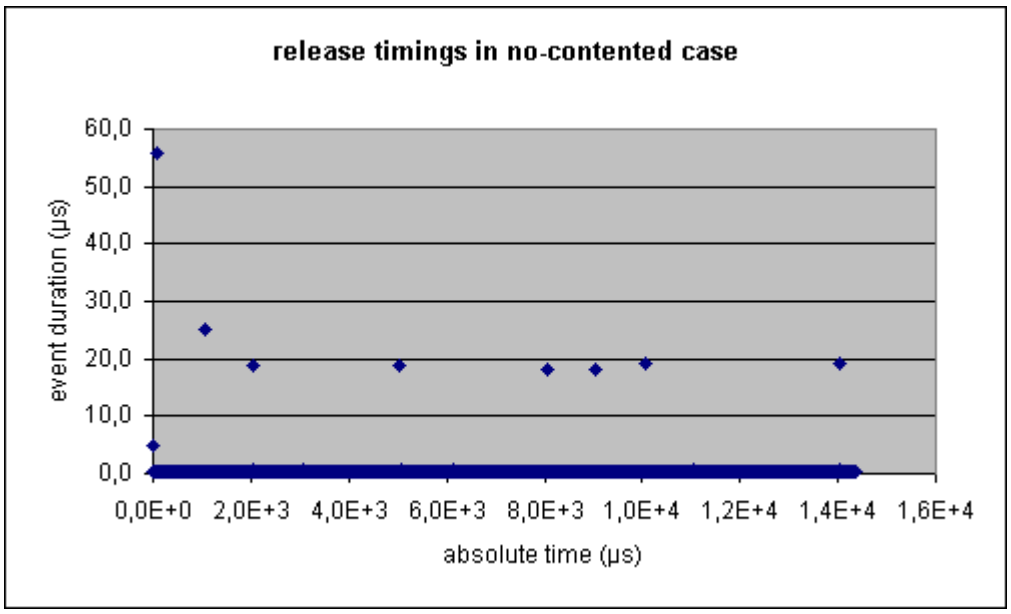
4.5.3.1 Test results

Test	result
Test succeeded	Yes

Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, no contention	7500	0.7 µs	20.8 µs	0.6 µs
Semaphore release time, no contention	7500	0.4 µs	55.8 µs	0.4 µs

4.5.3.2 Diagrams:





4.6 Interrupt tests (IRQ)

The performance of the interrupt handling in the operating system and hardware is tested here.

In a real-time system, interrupt handling is a major part of the system. Indeed, such systems 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 synchronised in any way with the platform under test.

Remark that we measure here the best case: we use a real interrupt while the other interrupts use `hardirq` thread. Adding an interrupt to thread switch latency will then give you interrupt latencies to the `hardirq` threads.

4.6.1 Interrupt latency (IRQ_P_LAT)

This test measures the time it takes to switch from a running thread to an interrupt handler. The time is measured from the moment the PCI interrupt line goes logically high (= electrical low).

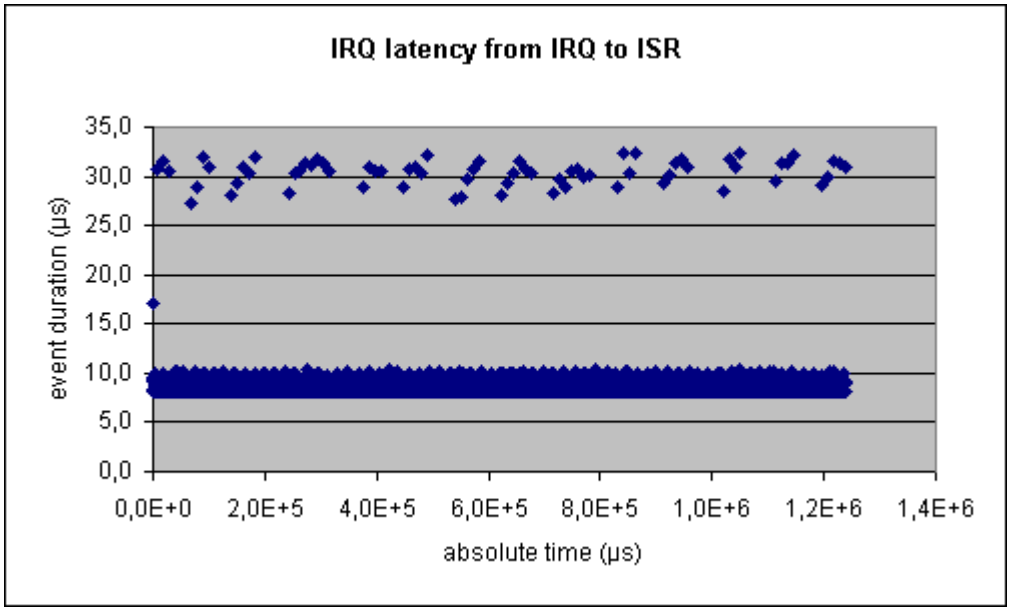
For the rest this looks pretty good, however this test supposes no load and a limited number of interrupts.

The clock time is easily detected again.

4.6.1.1 Test results

Test	Sample qty	Avg	Max	Min
Interrupt dispatch latency	10730	8.5 µs	32.4 µs	8.1 µs

4.6.1.2 Diagrams



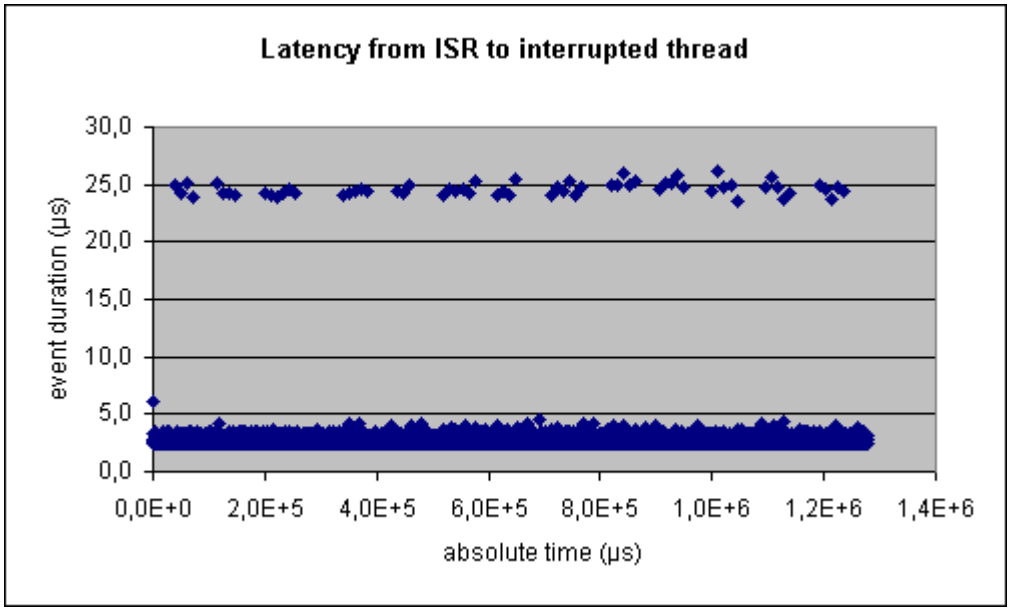
4.6.2 Interrupt dispatch latency (IRQ_P_DLT)

This measures the time it takes to switch from the interrupt handler back to the interrupted thread.

4.6.2.1 Test results

Test	Sample qty	Avg	Max	Min
Dispatch latency from interrupt handler	10732	2.7 µs	26.1 µs	2.4 µs

4.6.2.2 Diagrams



4.6.3 Interrupt to thread latency (IRQ_P_TLT)

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

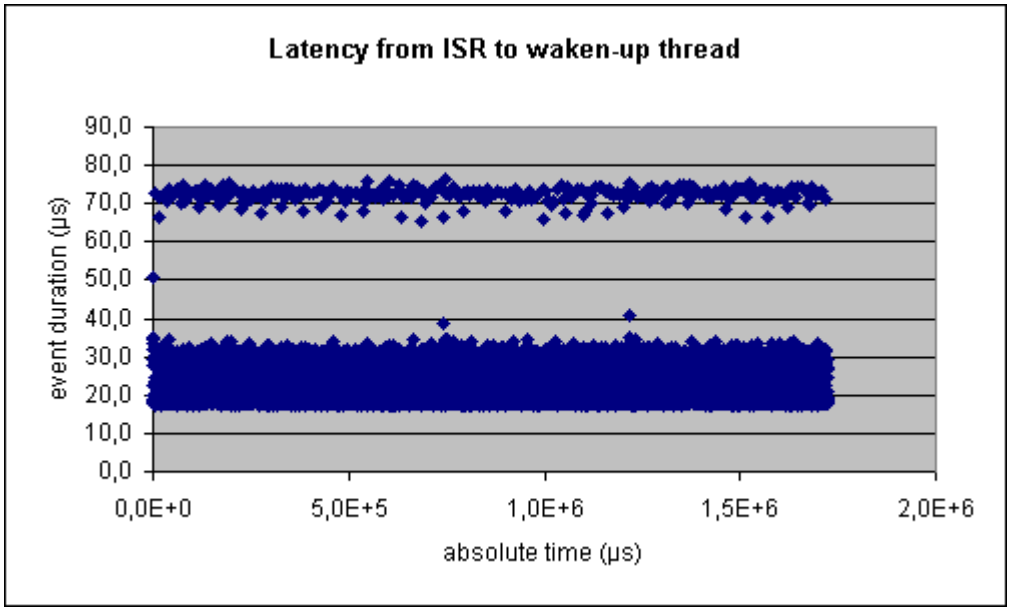
This test is done by allowing the interrupt handler to release a blocked thread. This blocking thread has the highest priority in the system and is blocked by an `ioctl` waiting for the next interrupt handled by our test device driver. There is also a low priority thread looping. So the measurement takes the time from the interrupt handler to the blocked thread (as a consequence this includes a thread switch).

In this case you see much more variation. Also the clock interrupt has its impact (in this case it seems to be always un-cached: 50µs).

4.6.3.1 Test results

Test	Sample qty	Avg	Max	Min
Latency from ISR to waken-up thread	15000	21.6 µs	76.4 µs	17.2 µs

4.6.3.2 Diagrams



4.6.4 Maximum sustained interrupt frequency (IRQ_S_SUS)

This test measures the probability an interrupt is missed. Is the interrupt handling duration stable and predictable?

The test is done on three levels:

- 100 interrupts, initial phase: a fast test just to see where we have to start searching.
- 1 000 000 interrupts, second phase based on the results from the first phase. This test still takes less than a minute and gives already accurate results.
- 1000 000 000 interrupts, takes more than 24 hours: to verify stability, therefore we cannot run a lot of tests, especially when it comes to large interrupt latencies.

As one can observe in the test results, although the interrupt latency is in the best case 8µs, the clock tick gives us a serious penalty here. On the long run, you can see that the guaranteed interrupt latency comes around 150µs. Depending on the drivers being used and loaded on the system, this latency can of course increase even more.

4.6.4.1 Test results

Interrupt period	#interrupts generated	#interrupts serviced	#interrupts lost
30 µs	1 000	963	37
40 µs	1 000	979	21
50 µs	1 000	994	6
60 µs	1 000	995	5
70 µs	1 000	997	3
80 µs	1 000	998	2
90 µs	1 000	998	2
100 µs	1 000	1 000	0
100 µs	1 000 000	1 000 000	0
100 µs	1 000 000 000	999 999 567	433
150 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***… etc.).

☺ The **Linux** under test passes this test well, no memory leaks were detected.

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

http://www.dedicated-systems.com
Email: info@dedicated-systems.com

5 Support

Support	0					NA					10

If you use downloaded open source software, you have only the internet as documentation resource available.

Although a lot of information can be found on the internet, concerning kernel configuration and RT_PREEMPT it is much more difficult to find adequate and complete documentation.

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