Behavior and performance evaluation of Android Linux 3.0.4 on ARM

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

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 authorization.
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
0.1	5 January 2012	All	Initial version
0.2	9-Jan-2012	All	Review the whole text
1.0	12-Jan-2012	ALL	Review text and add interrupts text
1.2	15 Feb 2012	All	Review again
2	17 Feb 2012	All	Change first page and some rephrasing of the tests
2.1	30 May 2012	Ratings section	There was an error in the ratings.

1 Document Intention

1.1 Purpose and scope

This document presents the quantitative evaluation results of the real-time **Linux** operating system, but as configured for Android, on an ARM-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" in 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

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 library used for interaction between the testing applications and the kernel, 3) 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 variants.

Recently, Google started with its Android phones and tablets. The advantage of Android is the availability of tools for making graphical user interfaces. All this is based on a Java virtual machine and thus hardware independent from the application level of view.

Now, what would happen if you would use for instance an Android system as a SCADA control system to control some machinery? The GUI would of course be a plus, but what would be the timing behavior?

Of course, Android is not meant to be used for any real-time purposes at all. It is even built to consume as less power as possible as it is used for handheld devices. We know that power management is an evil for good timing behavior! But even, we still wanted to compare it with a Linux RT_PREEMPT system on the same platform to clearly see the impact of kernel configuration choices made for Android and the ones used by the RT_PREEMPT patch.

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 operating system OS that will be evaluated is **Linux** 3.0.4 as used by the Linaro Android build for the Beagle XM platform. We used the following build: Linaro Android Beagle 11.09 release, build 4.

We deliberately did not adapt kernel configuration settings but used them as delivered.

2.2.1.1 Android

Android is based on Linux (as it uses a Linux kernel). However, a lot of customization on the libraries, root file system and available applications are done making an Android version much different than any traditional Linux system.

Remark that Android is not meant to be used for developing in C code. Everything is built around java. The Android development tools (Android SDK) are all focused to develop "smart apps" using the Java virtual machine. However, Android provides as well a Native Development Kit (Android NDK) which can be used to develop C/C++ applications.

But the Android NDK is not meant to be used for developing pure C/C++ applications. They are meant for developing libraries which provide some interfaces towards the java environment. So their purpose is to have some acceleration for dedicated algorithms instead. Thus, this puts already some constraints on what you can build natively for Android.

Another important difference relates to the C-libraries deployed with Android. Google wanted to isolate Android based applications from the GPL license. Therefore, neither the glibc nor the µClibc libraries can be used. Instead, they developed one themselves starting from the BSD code (using the BSD license, a truly free license as used for instance by FreeBSD and OpenBSD). In fact, Google started to build its C-libraries with a branch from the OpenBSD libraries source code. These C-libraries provided for the Android system are called the "Bionic Libc".

However, Bionic does not have all the features that are already in the traditional glibc implementation. Looking at the features required for real-time behavior, we notice the lack of priority inheritance mutexes. Although they are available in the kernel, you can access them only by building your own library above the Linux system calls…

Clearly, Android is intended to be deployed on tablets and smartphones, and not on other embedded usages. But we are still interested in the OS and that's why we test it here.

2.2.2 Hardware

The hardware that was used for executing our tests on the Linaro Android operating system was a Beagle-XM Board Rev C with following characteristics:

- based on the Texas Instruments DM3730 Digital Media Processor
- ARM Cortex A8 running at 1GHz
- L1 Cache: 32KB instruction and 32KB data cache
- L2 Cache: 64KB
- 512MB RAM at 166MHz

3 Evaluation results summary

Remember that the tested and evaluated product is an Android version, which is focused for smart phones and tablets. Thus, it is logic that it does not fit for real-time systems. However, due to the use of their proprietary bionic C library, it is even worse than expected!

First of all, Bionic does not have priority inversion protection mechanisms available on mutexes, which is an obligatory requirement to fulfill real-time requirements. Second, the semaphore implementation behaves badly: it is implemented as purely FIFO queued one (without any prioritization); further, release times are going straight through the roof once multiple threads are blocked on the same semaphore!

Finally, the bionic library contains only a limited subset of the glibc libraries. As such, a lot of open source C/C++ applications cannot be built for Android!

Conclusion is that Android should only be used where it is designed for: building Java GUI applications!

3.1 Positive points

- No license fees
- Source code available

3.2 Negative points

- No real-time characteristics at all!
- Not meant to be used for any C/C++ applications.
- **Bionic C library**, badly implemented semaphores and mutexes.

3.3 Ratings

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

RTOS Architecture (+libraries)	0	4	
OS Documentation	0	4	10
OS Configuration	0	6	10
Internet Components	0	6	10
Development Tools (C/C++)	0	6	10
Installation and BSP	0	4	10
Test Results	0	0	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	0									10

Our results shows that Android is completely inappropriate to fulfill any real-time requirements. It was not designed as such and thus it shouldn't be used for such systems.

Besides lacking priority inversion protection mechanisms and a badly behaving semaphore, it also has long duration clock ticks causing high interrupt latencies.

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)

As the Beagle board does not have any PCI support, we used the on-chip hardware timers for our measurements.

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.

For tracing, an internal General Purpose (GP) timer running at 13MHz was used. Reading out these timers takes some overhead of course; however, there is not any jitter at all in the overhead of the trace which in turn does not generate much extra inaccuracies.

Although it is possible to let the general purpose timer run at a higher frequency, the clock that was attached to this GP timer is also distributed to other components on the chip as well. Therefore, we had to stick with the configuration that was used by the OS on this board.

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	460 nsec
minimum tracing overhead	460 nsec
maximum tracing overhead	460 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.

Lately, we have been seriously reworking this test. 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 testing 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 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 organization:

- 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 the evaluated platform are shown below:

Test	no cache	cached	cache effect
CPU test: first load.	399.5 us		
CPU test: I Cache effect	230.1 us	31.5 us	7.3
MEM write test	29.1 us	21.1 us	1.4
MEM read test	44.2 us	21.2 us	2.1
Average caching effect (CPU and MEM)			3.6

The results show a behavior which is similar to the behavior of running Linux on our standard evaluation platform the Pentium MMX 200MHz:

- Instruction cache has the most impact on performance (you need to get each instruction from RAM, while there are always less than one memory accesses for each instruction). Remark that this test is built on purpose and thus an extreme worst case that in practice will never occur (you will never have >100KB instructions without loops in real environments).
- Write can be postponed, so it has less impact
- As read is blocking, it has more impact than the writes.

Because of the serious impact caused by not having your instructions in the cache, you should take extra safety margins in real-time behavior (worst case is less predictable) on this platform.

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.

As this Android based kernel is running at 128Hz, we expect a clock interrupt each 8ms. The following table shows the test results.

Test	result
Test succeeded	Yes
Tested clock period	8ms
Clock period adaptable	YES (by kernel config)

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.

☹ A worst clock time duration of about 300 µs is measured, which is extremely long! Clearly, each 100ms, the timer causes a major delay.

Further, taking a look at the clock ticking (skipping the long 300µs delay), we see that it is split up into three parts. This seems to be a typical phenomenon when using the low power 32 KHz OMAP timer to drive the operating systems clock tick.

Remark that in the evaluation of the Linux RT_PREEMPT on the same platform, a high frequency timer is used to drive the operating system clock. Android is built for portable devices, so it is logically optimized for low power consumption, which is typically a bad idea when you need real-time performance.

This shows clearly that you cannot build a solution that optimizes both! Life is always a trade-off. It all depends on what your priorities are.

4.2.2.1 Test results

Test	result
CLOCK_LOOP_COUNTER	10000
Normal busy loop time	42 µs
Busy loop time with clock interrupt	70 µs, worst case 350 µs
Clock interrupt duration	Around 30µs, with each 100ms a spike of around 300µs.

4.2.2.2 Diagrams

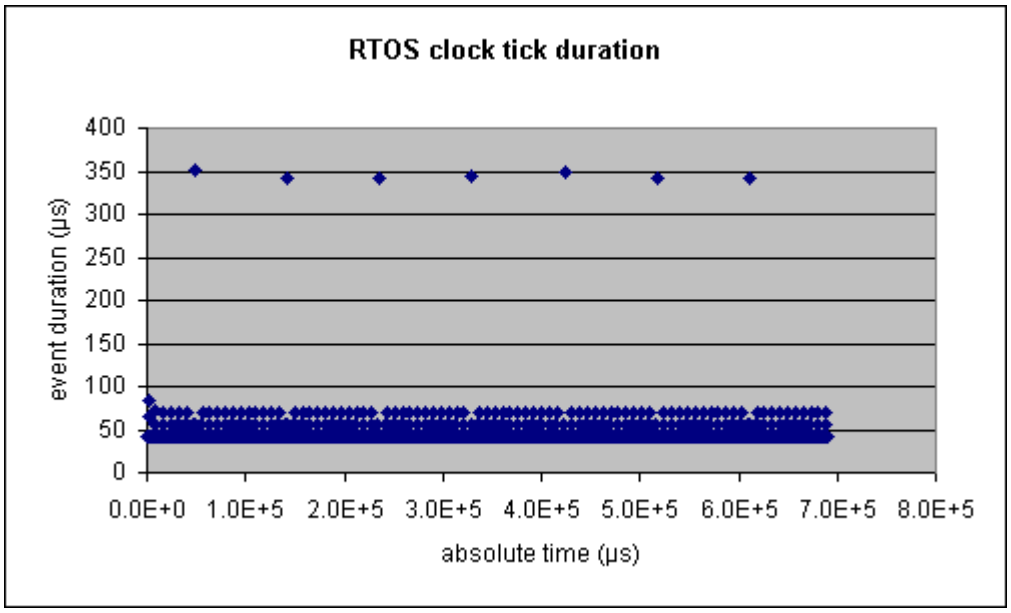


Figure 1: RTOS clock tick duration (1)

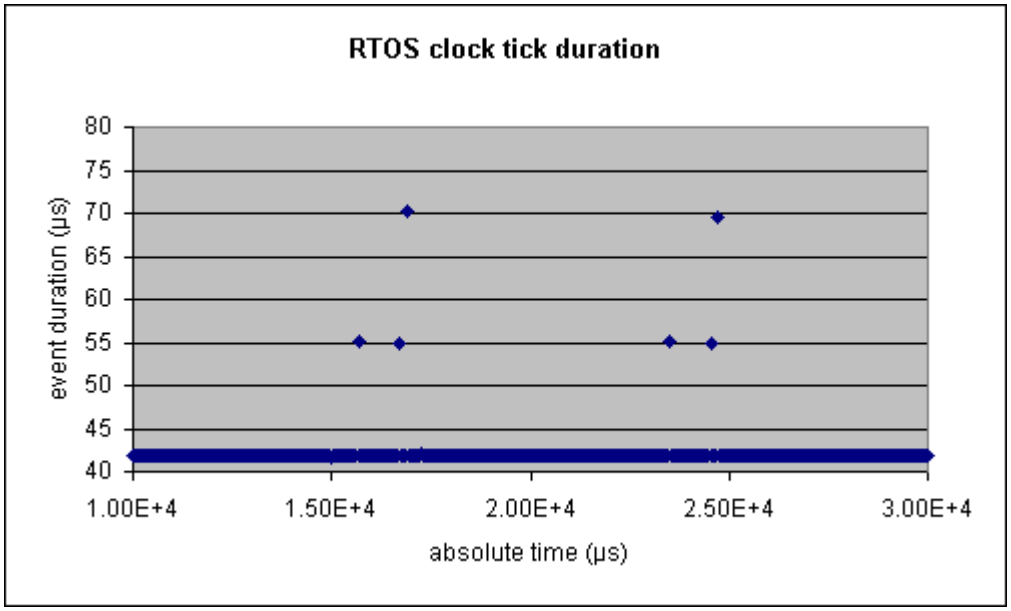


Figure 2: RTOS clock tick duration, zoomed in view (2)

4.3 Thread tests (THR)

Thread tests measure the scheduler performance.

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

- The queuing behavior 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 around 5ms, which is about 15 times longer than the other cases.

This is a Linux problem which is also present with the RT_PREEMPT patch.

As we were using the standard Android Linux, the "run-away protection" for real-time threads was still enabled. So after the real-time threads were active for 0.95 seconds, the kernel scheduler will give a period of 5ms to the non-real-time threads. Remark that, this can be configured by adapting the "`/proc/sys/kernel/sched_rt_runtime_us`" kernel parameter. Its default value is 95000. To disable this protection, just set it to -1. In the rest of the tests performed, this run away protection was disabled.

In real-time design, it is a bad practice to dynamically create and terminate threads, and to 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)

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?

This test succeeded.

However, we observed different behaviors depending on whether SCHED_FIFO or the SCHED_RR class was used. When lowering the priority of a thread, then this thread:

- is placed at the head of the ready queue if the Linux OS is running with SCHED_RR policy
- is placed at the end of the ready queue if the Linux OS is running with SCHED_FIFO policy

Note 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)

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 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 10Hz frequency.

4.3.2.1 Test results

Test	result
Test succeeded	NO: first yield upon thread creation uses 1000ms and next yields take 93ms!
RR Time slice following this test	100 ms

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).

Further you will see clearly the influence of clock interrupts (causing the maximum values in the graphics). In the 1000 threads tests, we were lucky not to catch the 300µs delay.

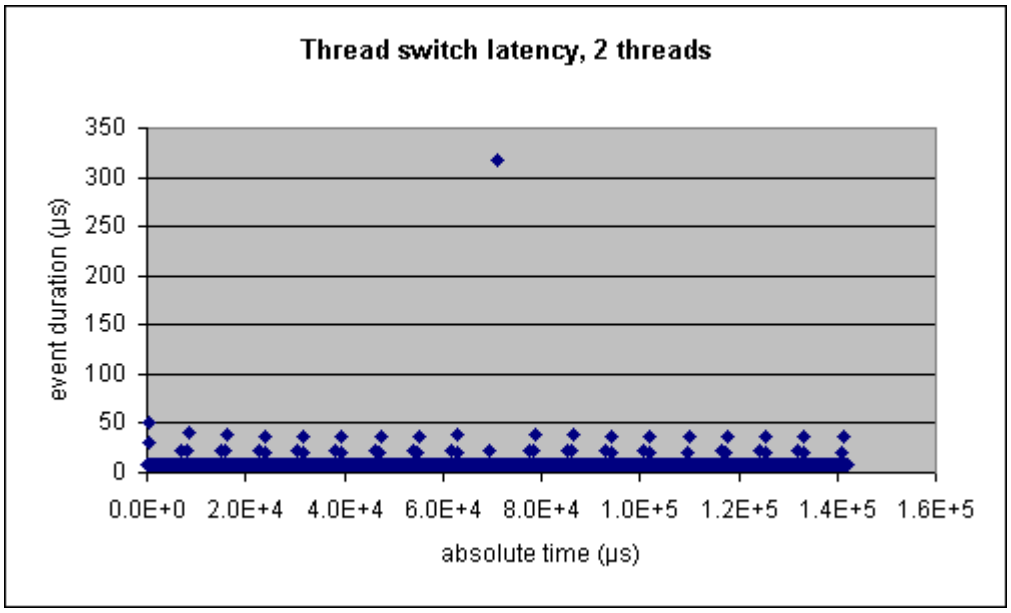
Besides the clock interrupts, the thread switch latency is a fairly stable line (figures below), which is good.

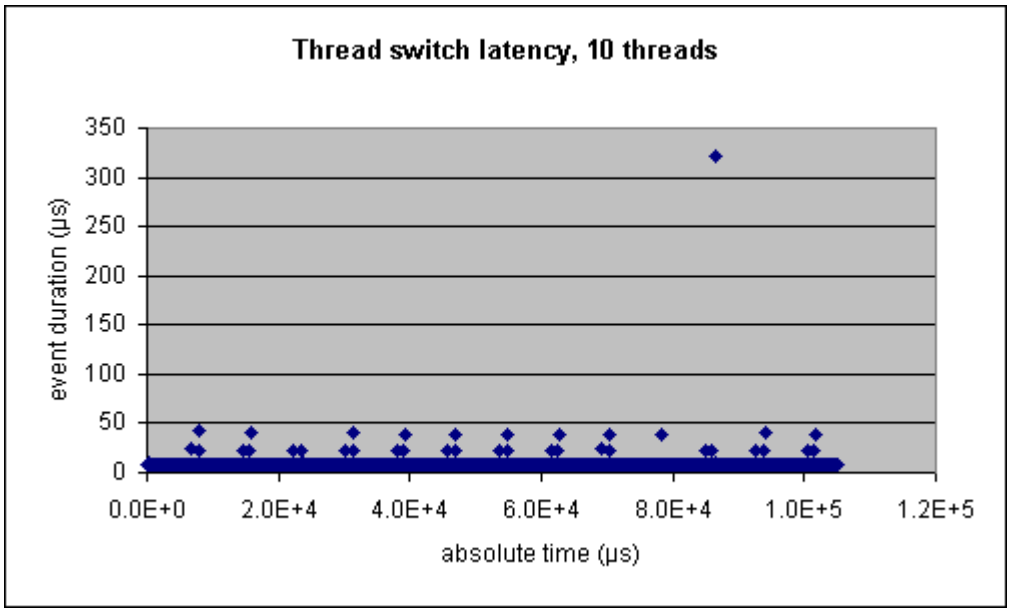
4.3.3.1 Test results

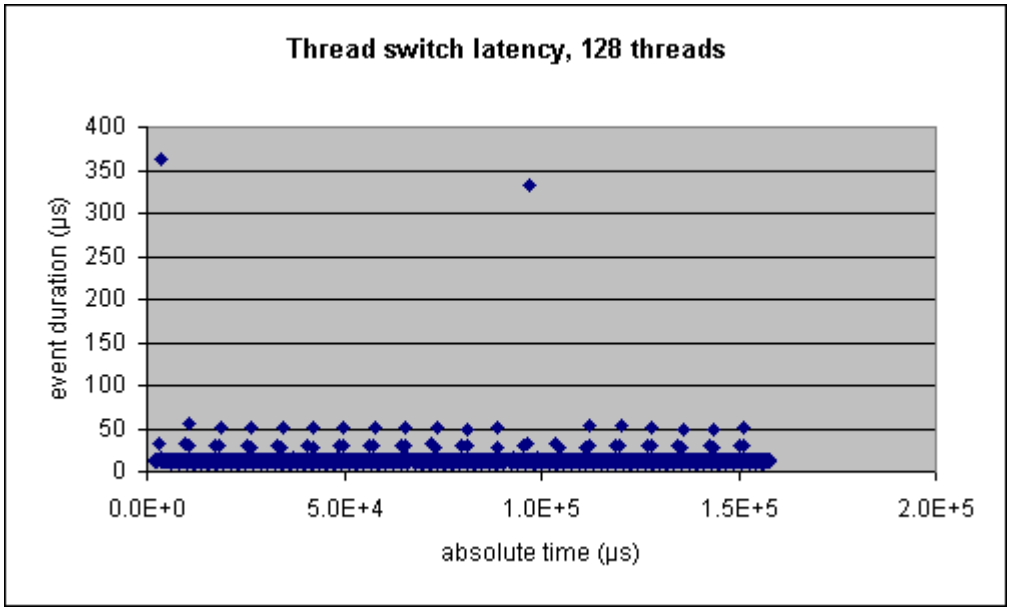
Test	result
Test succeeded	YES

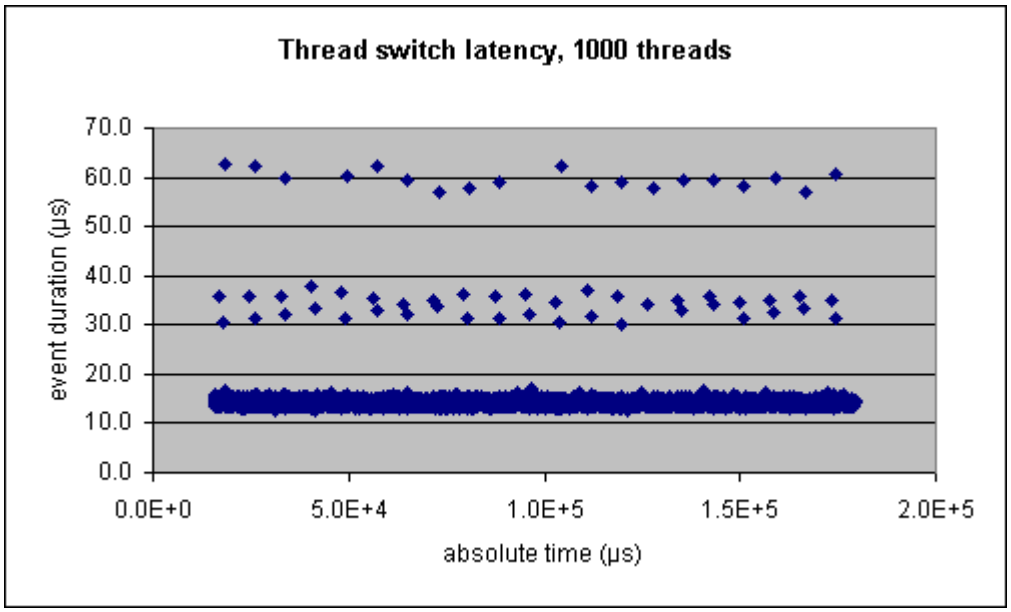
Test	Sample qty	Avg	Max	Min
Thread switch latency, 2 threads	15999	7.9 µs	317 µs	7.6 µs
Thread switch latency, 10 threads	10664	8.4 µs	321 µs	7.9 µs
Thread switch latency, 128 threads	10624	13.2 µs	363 µs	9.5 µs
Thread switch latency, 1000 threads	10334	14.3 µs	62.8 µs	12.8 µs

4.3.3.2 Diagrams









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, 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.

Android, which uses its own Bionic C library, behaves here very different compared with the glibc library. Both creation and deletion of threads is deferred for some management system. For instance, in the case where threads are created/deleted without any chance to run (scenario 1), the thread will actually never become active; it will never exist in the Linux kernel. The creation in the Linux kernel will happen when it actually becomes active. It is good that this system still preserve priorities on creation, which is not easy to handle in such cases.

Also deleting threads is deferred, but this is done by some manager thread that is not running at the same priority of the thread being deleted. As a result, we could not run our scenario 3 test.

Again, this shows us that the behavior depends not only on the kernel, but also on the used libraries as well!

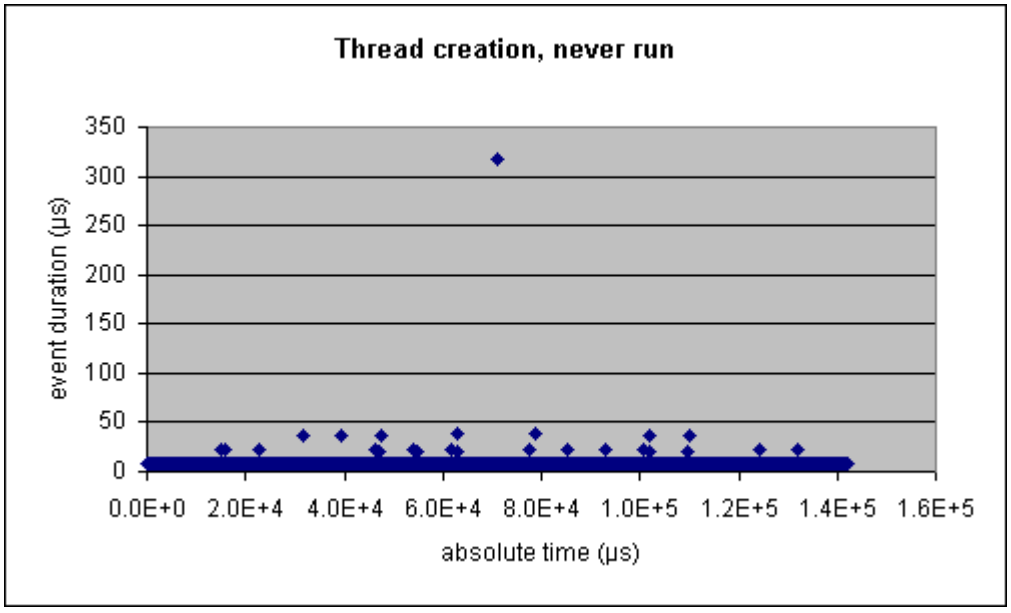
As expected the 300µs clock tick duration sometimes interferes with our tests…

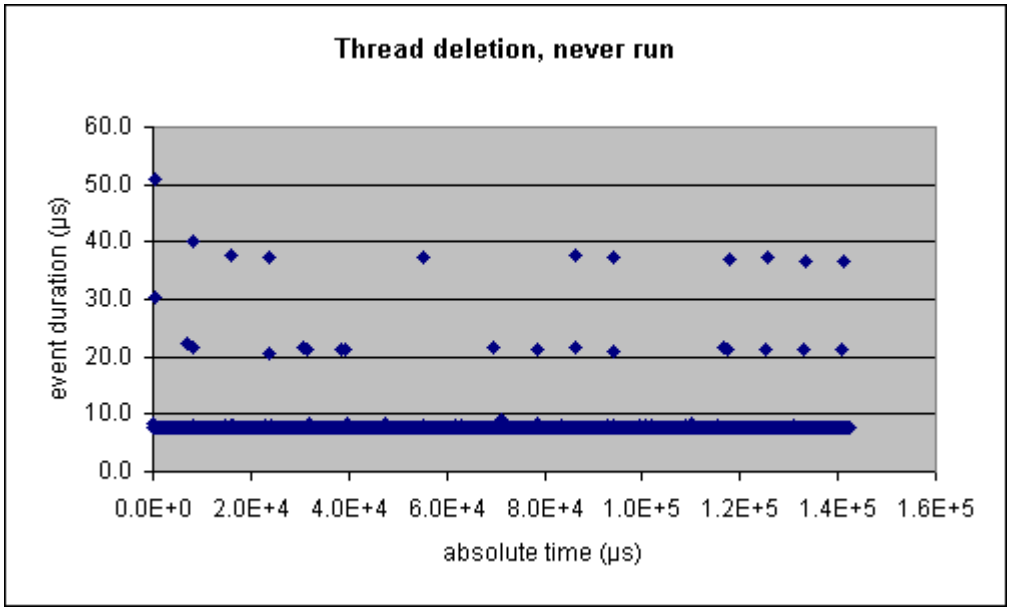
4.3.4.1 Test results

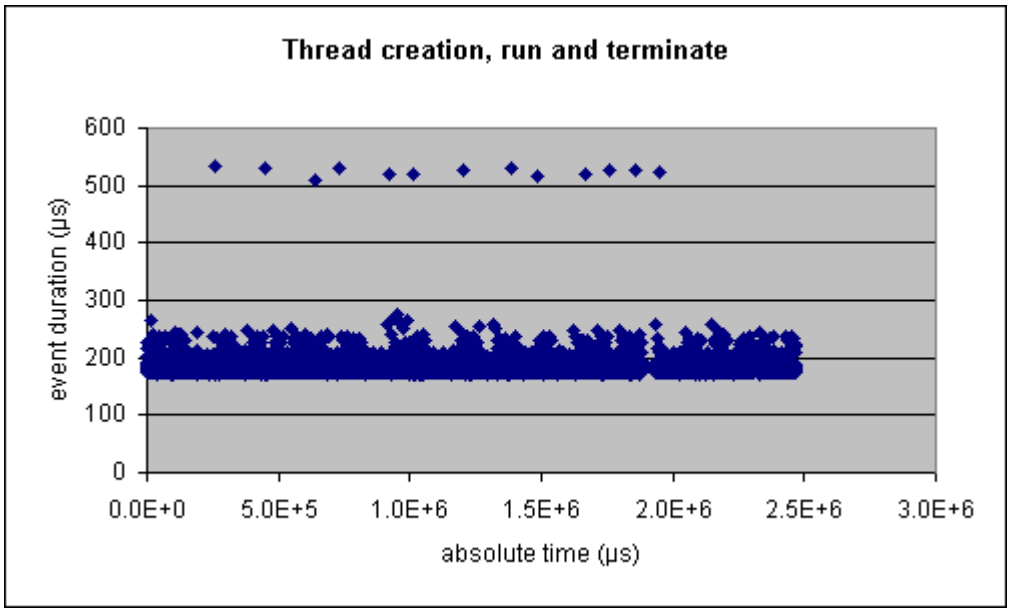
Test	result
Test succeeded	YES

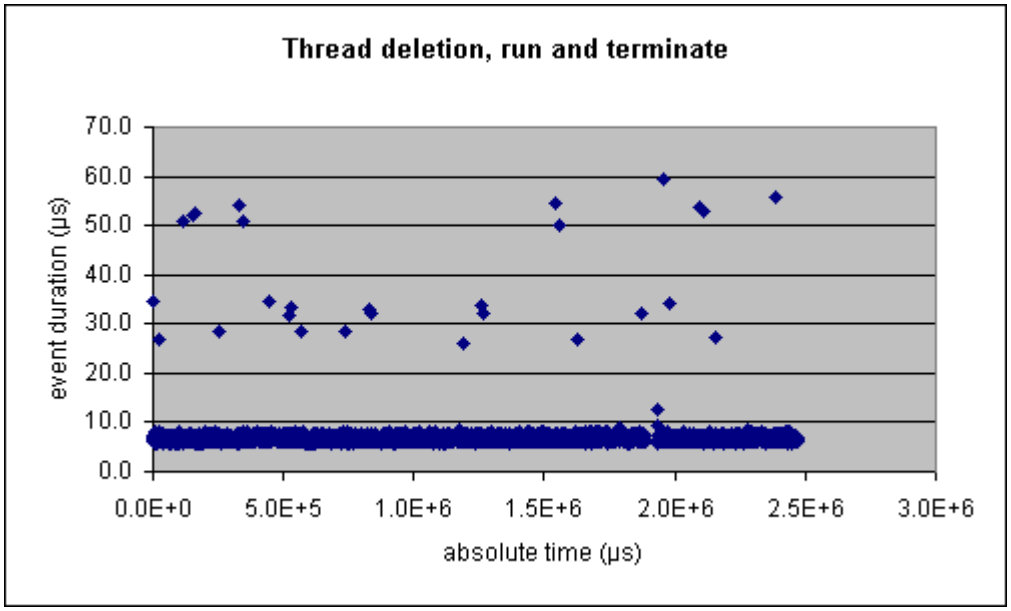
Test	Sample qty	Avg	Max	Min
Thread creation, never run	7999	7.9 µs	317 µs	7.7 µs
Thread deletion, never run	7500	7.8 µs	51.1 µs	7.6 µs
Thread creation, run and terminate	7500	183.4 µs	532 µs	169.7 µs
Thread deletion, run and terminate	7500	6.8 µs	594 µs	5.5 µs
Thread creation, run and block	7500	NA	NA	NA
Thread deletion, run and block	7500	NA	NA	NA

4.3.4.2 Diagrams









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

☹ **Android did not pass this test**! This shows us again the importance of the libraries… The Bionic C libraries use an un-prioritized FIFO on a semaphore lock. So the thread which is blocked the longest will be activated, not the one with the highest priority!

Clearly, this breaks all rules for good real-time design. Keep in mind that Android is made to run Java applications and it is clearly not foreseen to run native applications.

When real-time priorities are not used, then Linux priorities (as seen in the kernel) are not strict. Priority will be used only to assign a larger amount of the available CPU cycles for some threads more than others. So for non-real-time threads such a semaphore, implementation is of course not an issue.

4.4.2.1 Test results

Test	result
Test succeeded	NO

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.

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.

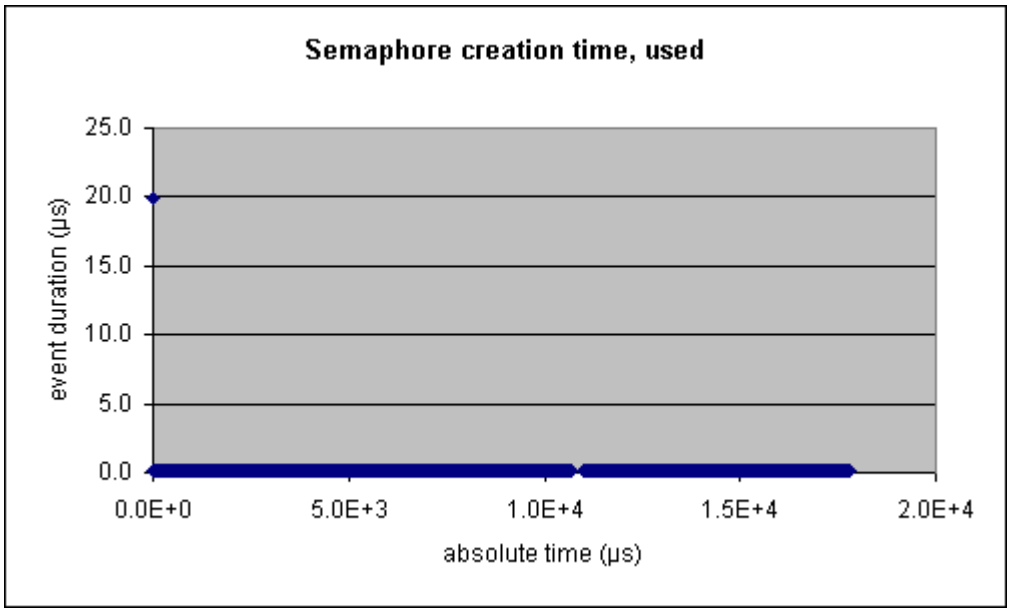
Remark that we do not use "named" ***semaphores***, so they cannot be used between processes. Therefore, no access to the kernel is required to create them.

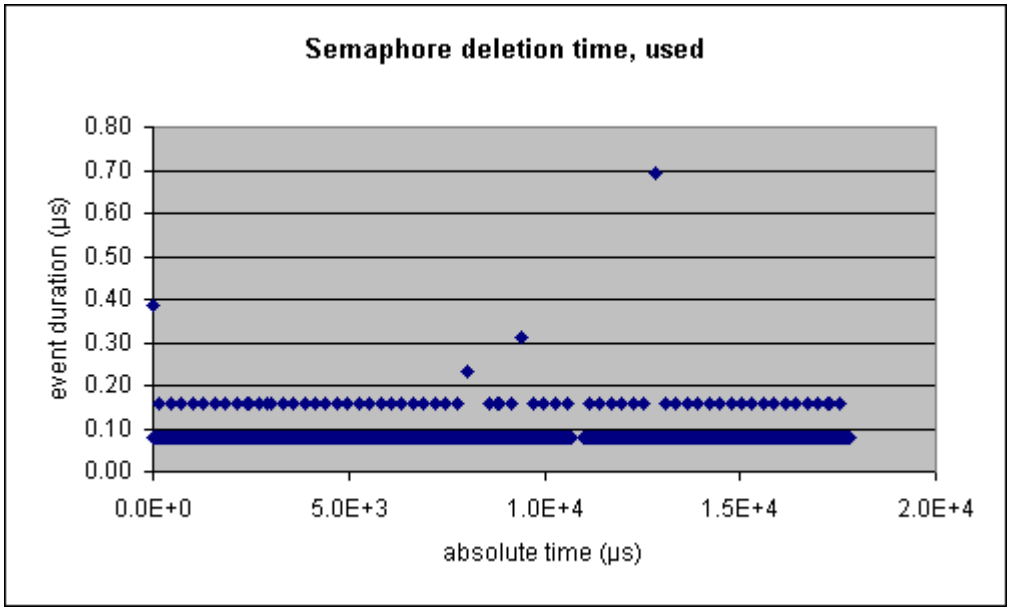
4.4.3.1 Test results

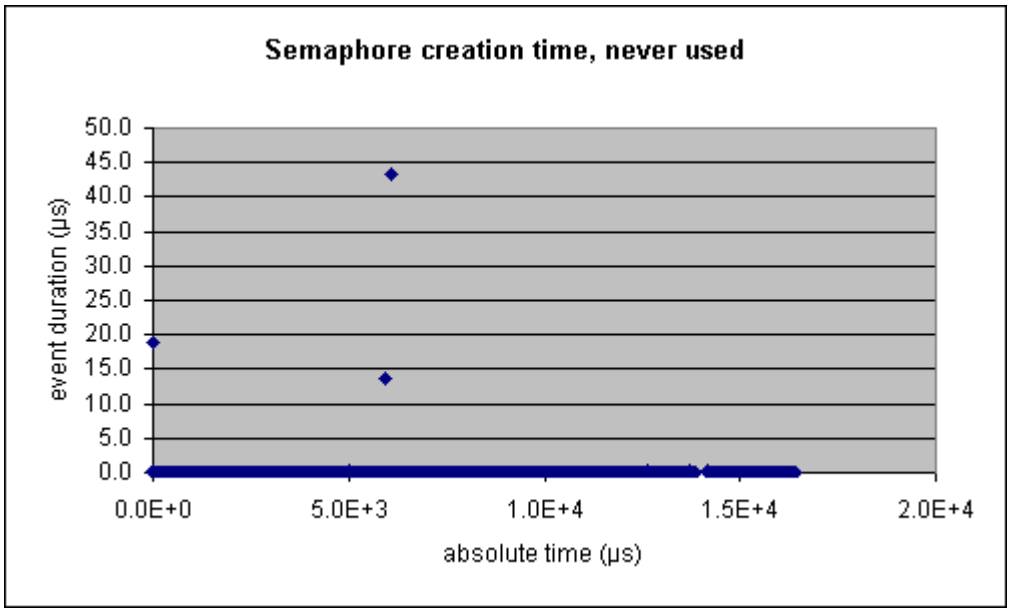
Test	result
Test succeeded	YES

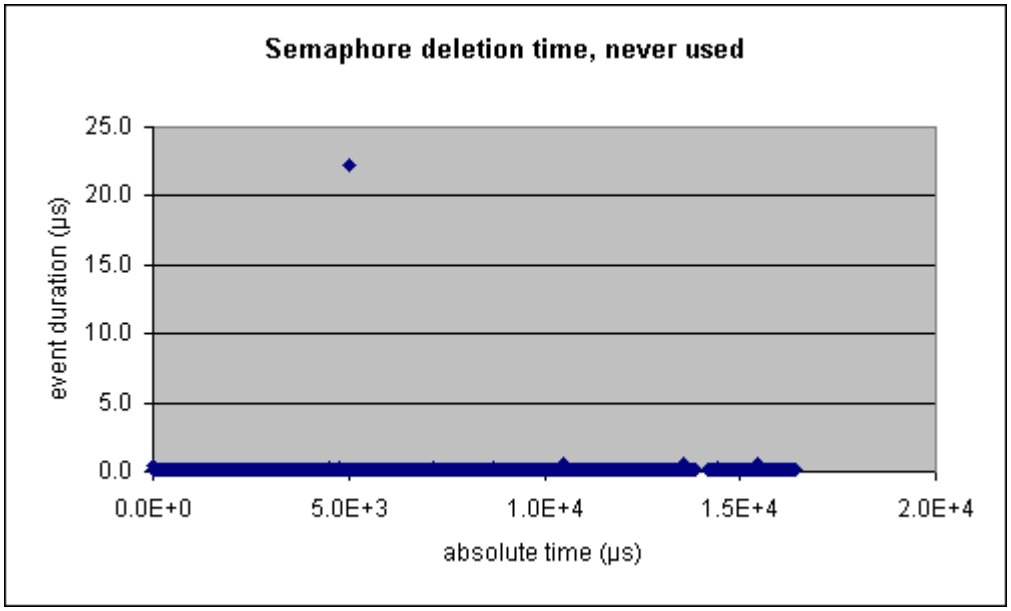
Test	Sample qty	Avg	Max	Min
Semaphore creation time, used	8000	<0.1 µs	19.9 µs	<0.1 µs
Semaphore deletion time, used	8000	<0.1 µs	0.7 µs	<0.1 µs
Semaphore creation time, never used	8000	<0.1 µs	43.2 µs	<0.1 µs
Semaphore deletion time, never used	8000	<0.1 µs	22.3 µs	<0.1 µs

4.4.3.2 Diagrams









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.

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.

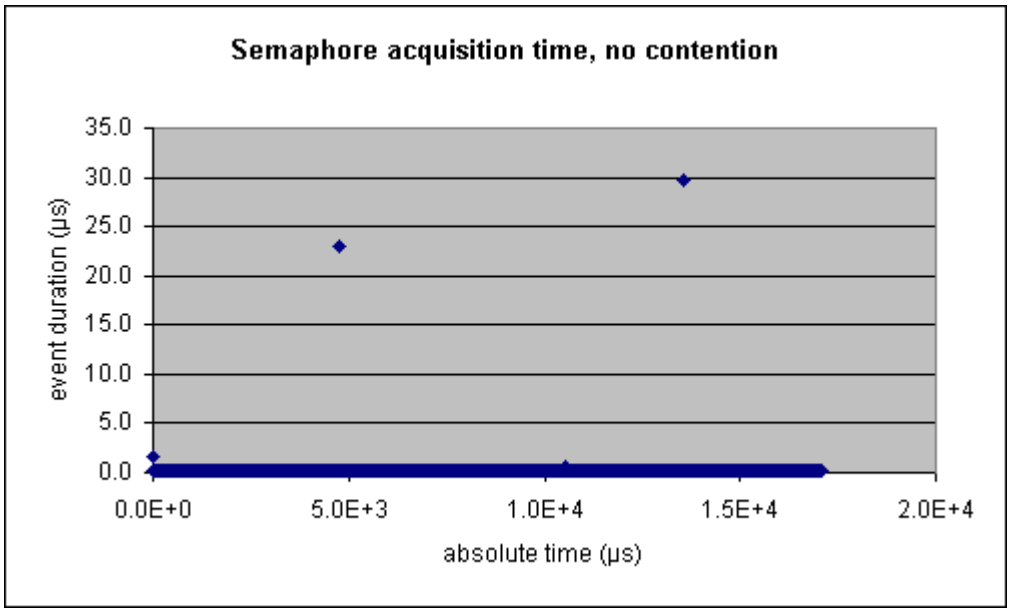
The clock tick spike is always present.

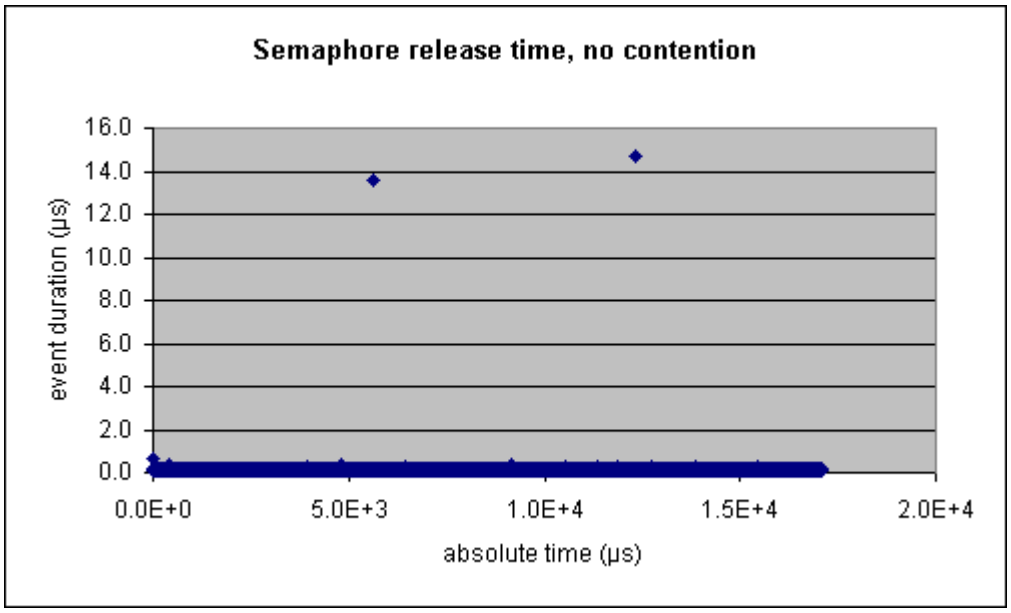
4.4.4.1 Test results

Test	result
Test succeeded	YES

Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, no contention	8000	0.1 µs	29.6 µs	<0.1 µs
Semaphore release time, no contention	8000	0.1 µs	14.7 µs	<0.1 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 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 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.
- After the last thread is created and 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...

☹ This test completely failed on Android! As you can see on the semaphore release extract of one test loop, the number of threads pending on the semaphore drastically impact the time needed to release it! If only one thread is pending on it, the release time takes an acceptable 15µs. But with 90 threads pending on it, this reach a 30 times higher figure: around 450µs!

So the Bionic C library implementation seems to have an extremely bad behaving semaphore. First, it is not priority based but FIFO based. Second, its release times are extremely depending on the number of pending threads being blocked on the semaphore.

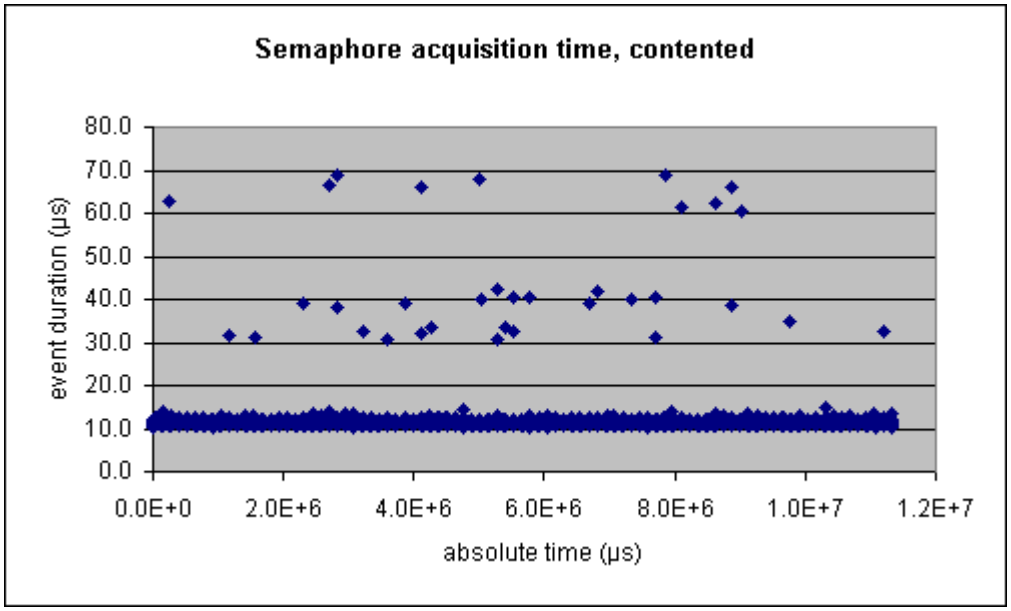
It is strange that a simple FIFO implementation can still behave so badly upon semaphore release!

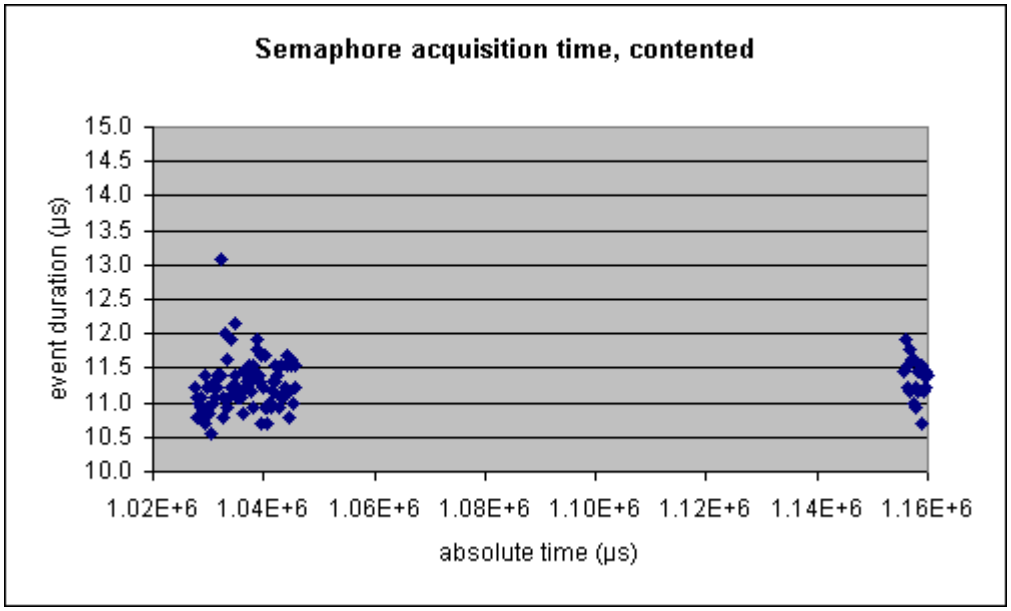
4.4.5.1 Test results

Test	result
Test succeeded	NO
Max number of threads pending	90 (close to the number of priority levels)

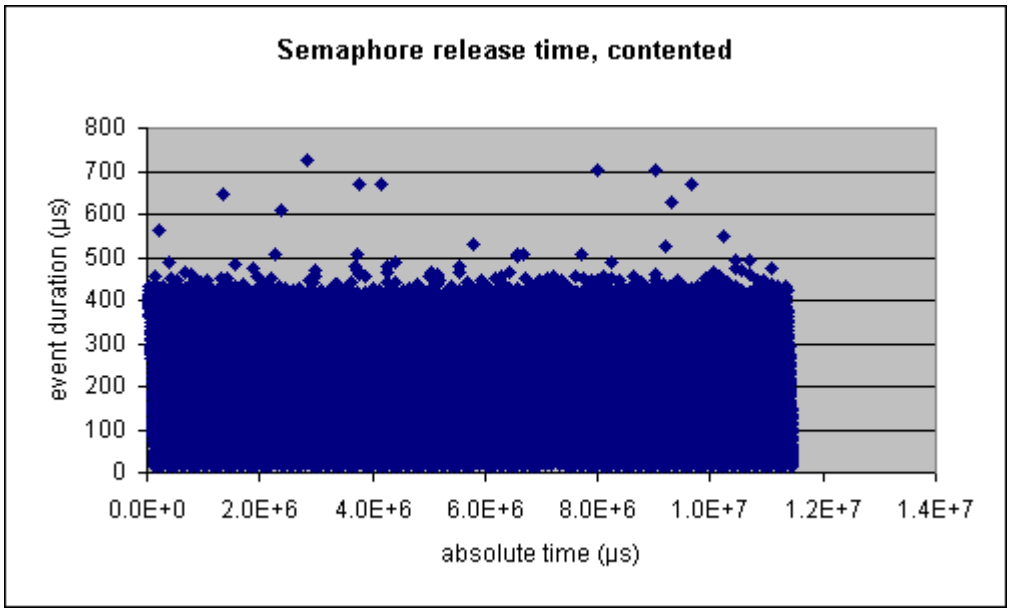
Test	Sample qty	Avg	Max	Min
Semaphore acquisition time, contented	7921	11.5 µs	68.9 µs	10.4 µs
Semaphore release time, contented	7921	211 µs	727 µs	14.5 µs

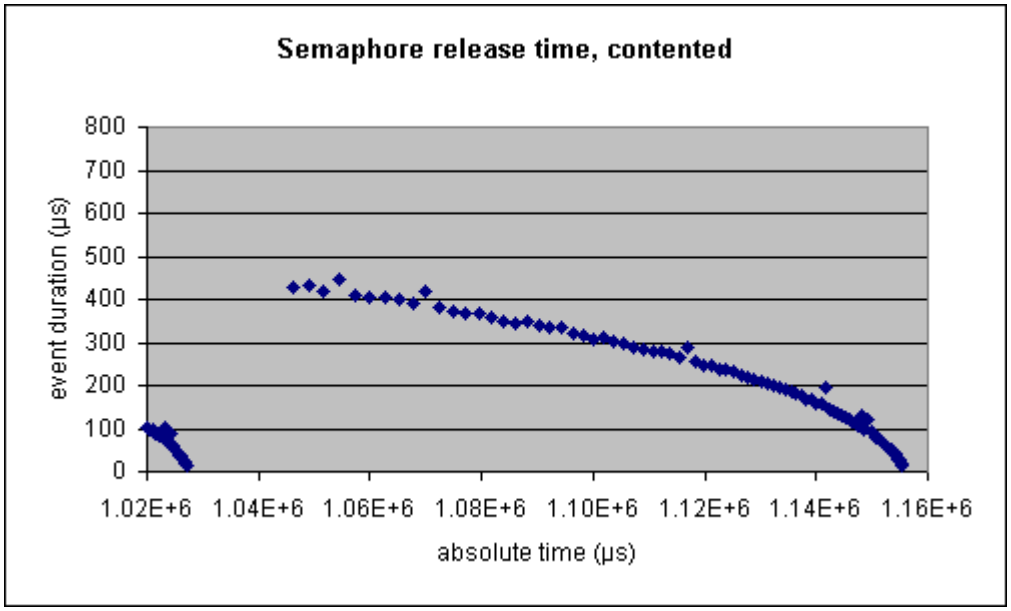
4.4.5.2 Diagrams





Extract of one test loop iteration





Extract of one test loop iteration

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.

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.

☹ **Android did not pass this test**!

Although the standard Linux kernel has support for this, the Bionic C libraries do not export such functionality. So there is nothing present to avoid priority inversions!

4.5.1.1 Test results

Test	result
Priority inversion avoidance system call present	NO
System call used	pthread_mutex_lock
Test succeeded	NO
Priority inversion avoided	NO
Mechanism used if any?	There is none!

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).

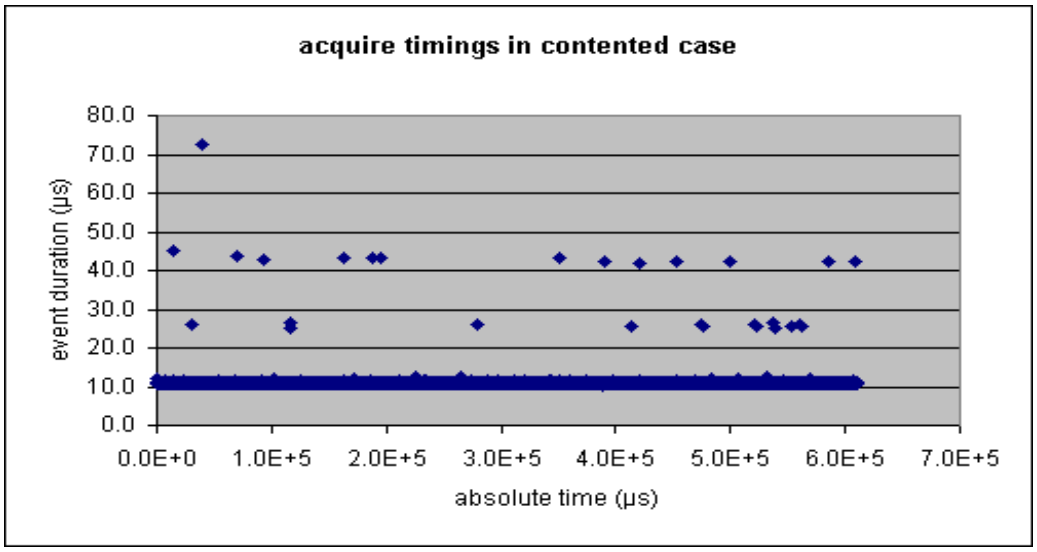
We measured the release time from the release call to the moment the thread requesting the mutex was activated; so this measurement also includes a thread switch.

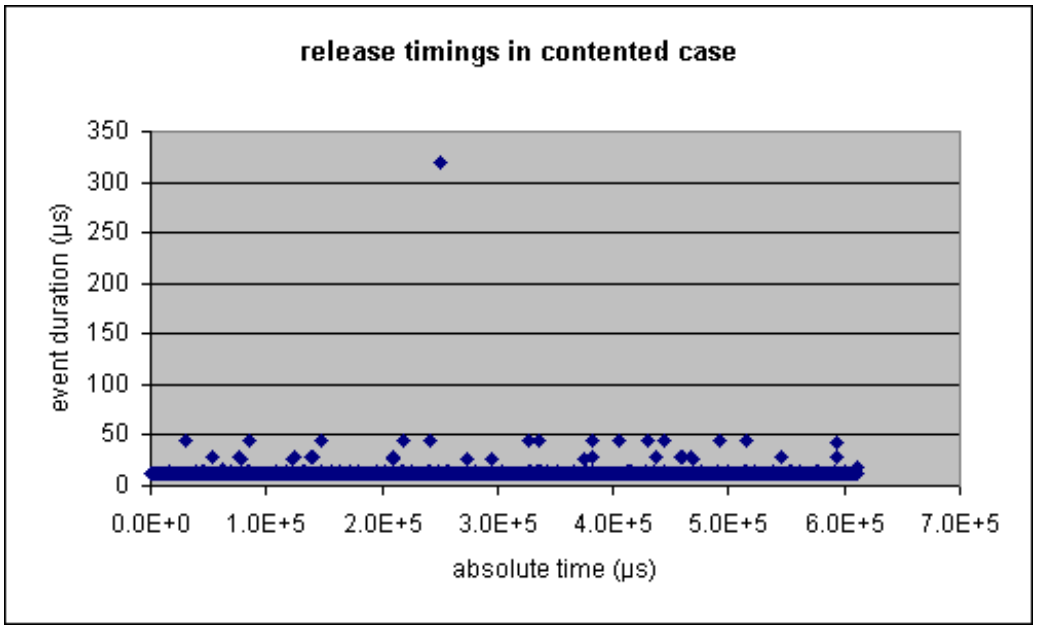
4.5.2.1 Test results

Test	result
Test succeeded	Yes

Test	Sample qty	Avg	Max	Min
Mutex acquisition time, contention	8000	11.0 µs	72.6 µs	10.5 µs
Mutex release time, contention	8000	12.8 µs	320 µs	12.3 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. Note that the required speed is only possible if the CPU supports some type of atomic instruction, so that no system call is needed when no contention is detected.

As expected, this doesn't take a lot of time; it is in fact too small to be measured.

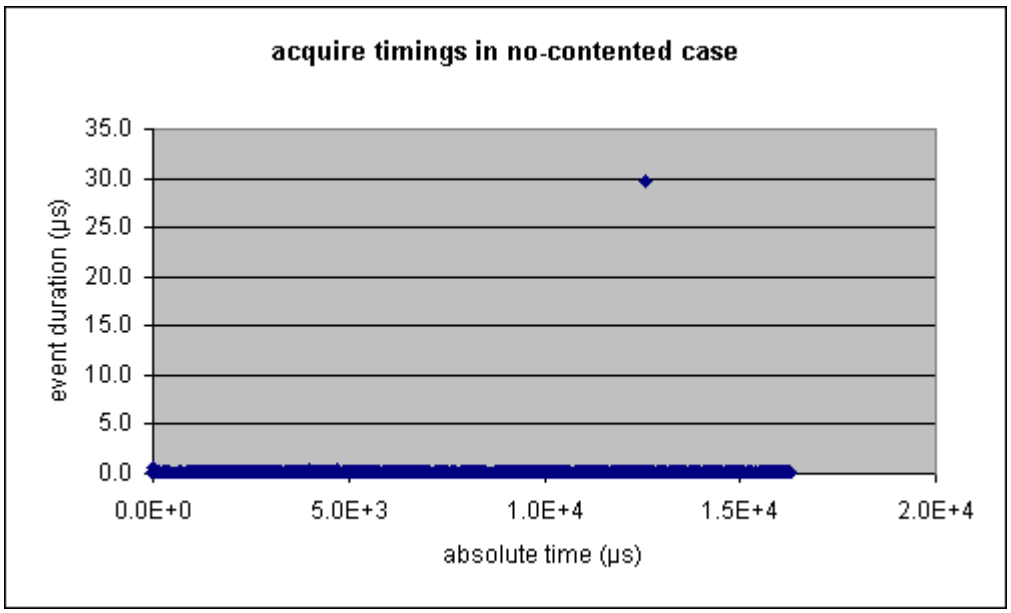
Remark that the timer used for the measurements is running at 13MHz. Therefore, you can see here the resolution of about 0.077 µs.

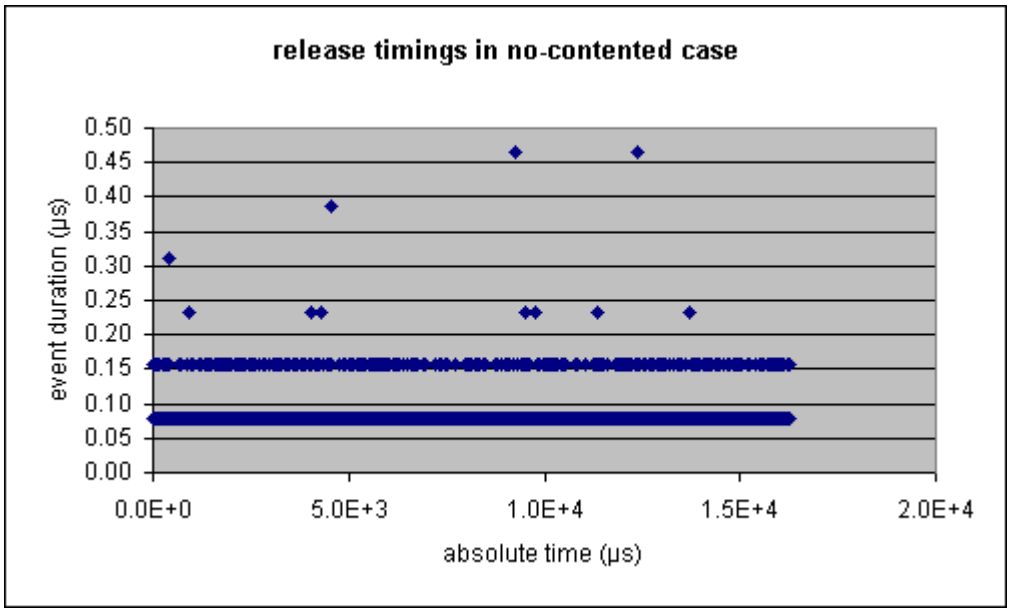
4.5.3.1 Test results

Test	result
Test succeeded	Yes

Test	Sample qty	Avg	Max	Min
Mutex acquisition time, no contention	8000	<0.1 µs	296 µs	<0.1 µs
Mutex release time, no contention	8000	<0.1 µs	0.5 µs	<0.1 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 our interrupt tests, we use a general purpose timer on the BeagleBoard-XM chip to generate interrupts, in the same way that we use a general purpose timer on the chip for tracing. The timer we used has an independent programmable wrap-around timer, which protects it from influence by the RTOS clock. This protection allows us to guarantee that an independent interrupt source is not synchronized in any way with the platform being tested.

4.6.1 Interrupt latency (IRQ_P_LAT)

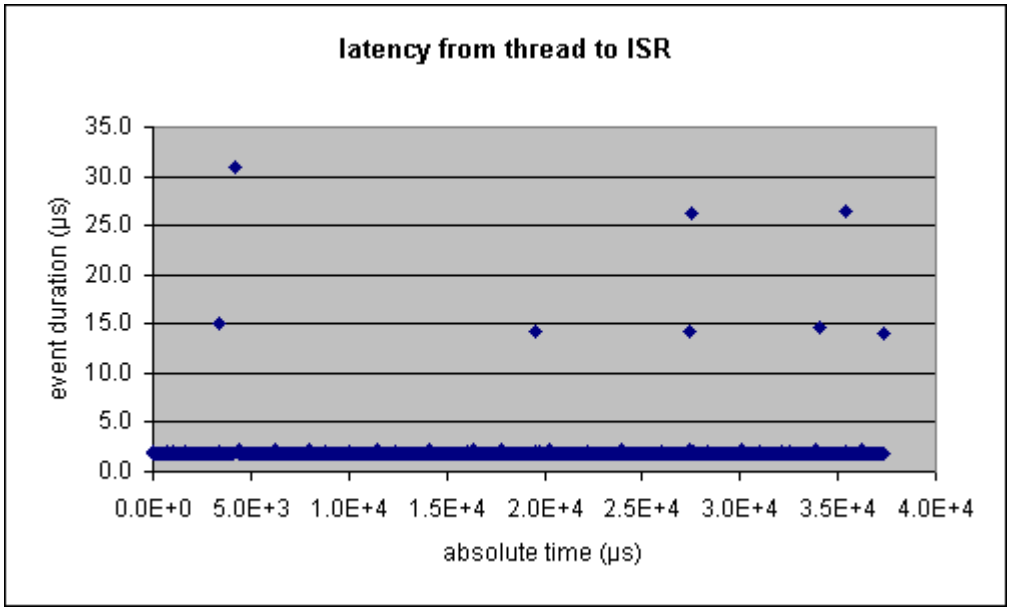
The "interrupt latency" test measures the time it takes to switch from a running thread to an interrupt handler. This time is measured from the moment the running thread is interrupted, so the measurement does not take into account the hardware interrupt latency.

The clock time is easily detected again.

4.6.1.1 Test results

Test	Sample qty	Avg	Max	Min
Interrupt dispatch latency	1605	1.9 µs	30.9 µs	1.8 µs

4.6.1.2 Diagrams



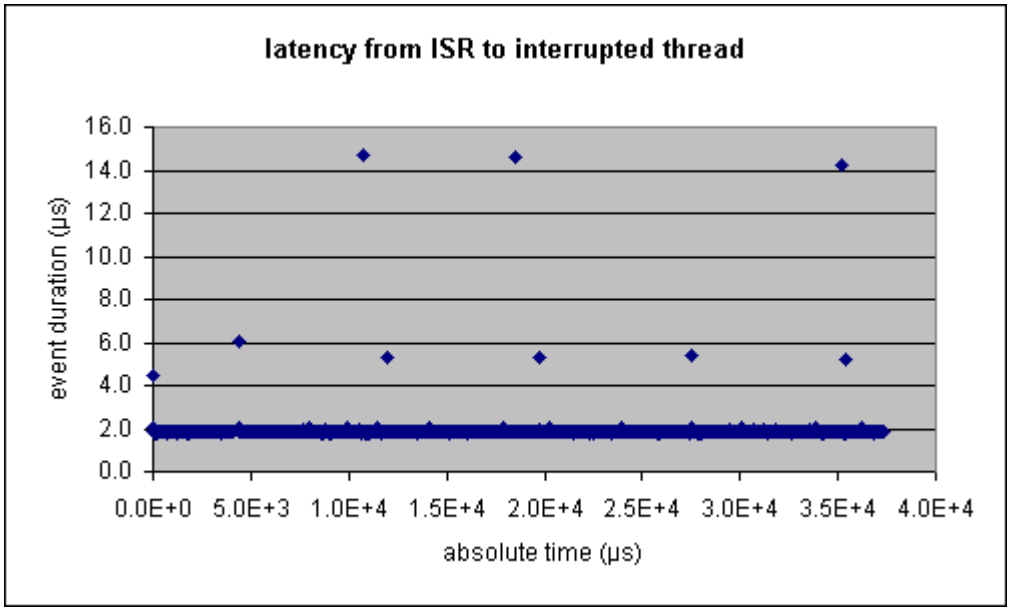
4.6.2 Interrupt dispatch latency (IRQ_P_DLT)

The “interrupt dispatch latency” test measures the time the OS 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	1605	1.9 µs	14.7 µs	1.8 µs

4.6.2.2 Diagrams



4.6.3 Interrupt to thread latency (IRQ_P_TLT)

The "interrupt to thread latency" test measures the time the OS takes to switch from the interrupt handler to the thread that is activated from the interrupt handler.

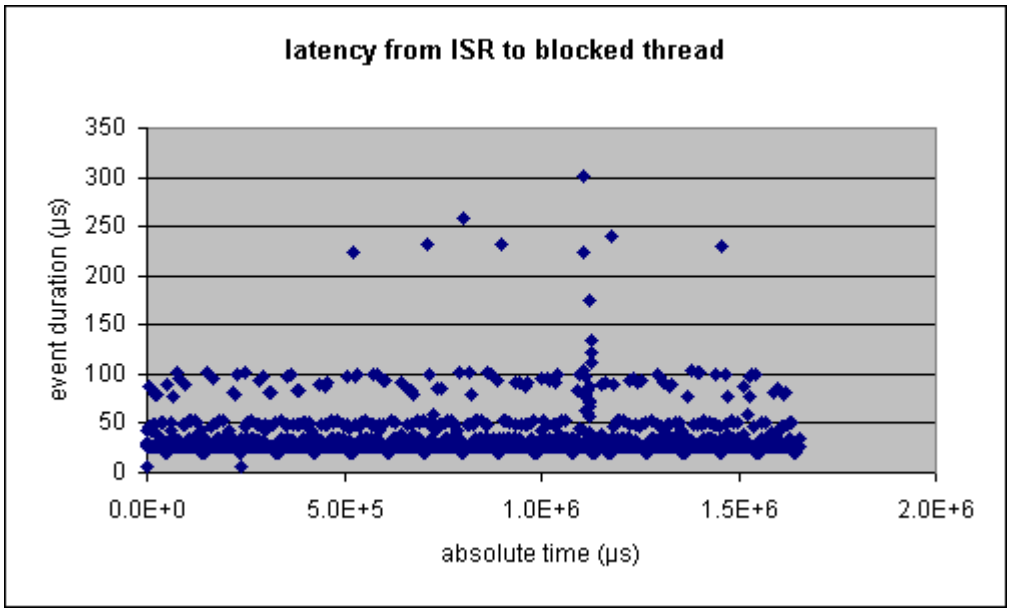
This test is done by allowing the interrupt handler to release a blocked thread. This blocking thread has the highest priority in the system and is blocked by an `ioctl` waiting for the next interrupt handled by our test device driver. There is also a low priority thread looping. So the measurement takes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

In this case, you see large variations. Of course the long clock interrupt has its impact (up to 300µs), but even without it, the release time is very long; Note that using Linux RT_PREEMPT, the worst case is around 13µs on the same platform…

4.6.3.1 Test results

Test	Sample qty	Avg	Max	Min
Latency from ISR to waken-up thread	3995	29.5 µs	300 µs	5.2 µs

4.6.3.2 Diagrams



4.6.4 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.
- 1 billion 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. Normally we do this on a billion interrupts, but due to the long interrupt duration, we were not able to perform the one billion interrupt test. Instead we had to stick to 10 million interrupts (otherwise the test starts to take too long).

As one can observe in the test results, although the interrupt latency is in the best case 100µs, the clock tick gives us a serious penalty here. On the long run, you can see that the guaranteed interrupt latency comes around 410µs.

4.6.4.1 Test results

Interrupt period	#interrupts generated	#interrupts lost
100 µs	10 000	23
120 µs	10 000	17
150 µs	10 000	2
180 µs	10 000	0
310 µs	100 000	3
330 µs	100 000	2
350 µs	100 000	0
350 µs	1 000 000	4
370 µs	1 000 000	0
370 µs	10 000 000	6
390 µs	10 000 000	3
410 µs	**10 000 000**	**0**

5 Support

Support	0					NA					10

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

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