Comparison of QNX Neutrino 6.5 and RT Linux on X86 (MMX)

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

1 About the RTOS evaluation project

This section describes the purpose and scope of the evaluations conducted by Dedicated Systems.

1.1 *Purpose and scope of the RTOS evaluation*

This document provides quantitative measures to help potential RTOS users make objective comparisons between OSs, and help them decide which OS is better for their needs. This document compares the results of the quantitative evaluations of QNX Neutrino 6.5 and RT Linux 2.6.33.7 real time operating systems (RTOSs).

Both RTOSs were evaluated on the same x86 platform (Intel Pentium MMX).

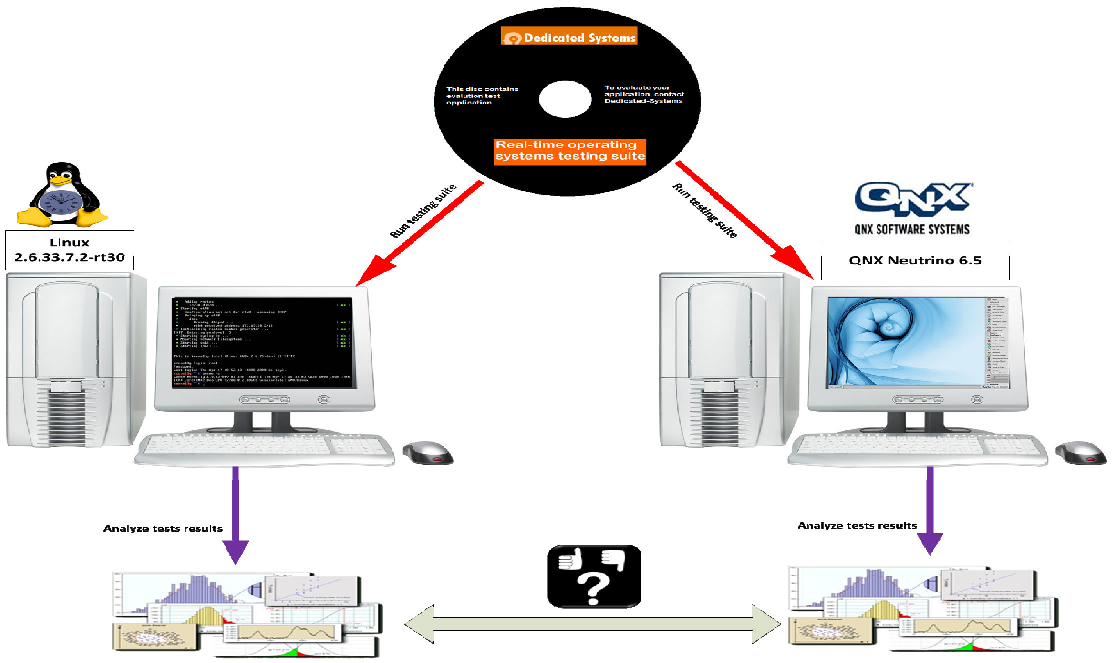


Figure 1: High level view of the evaluation procedure

1.2 *Test framework used: 2.9*

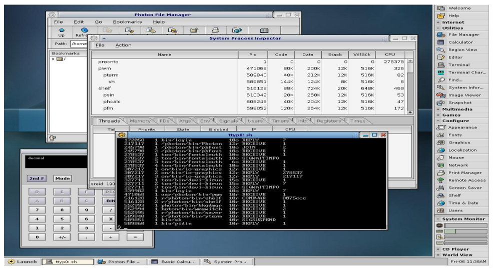
This document shows the test results in the scope of the evaluation framework 2.9. More details about this framework are found in Doc 1 (see section 6).

2 About the OSs and the testing platform

This section describes the OSs that Dedicated Systems tested using its Evaluation Testing Suite, and the hardware on which these OSs were running during the testing.

2.1 Software

The following table shows the operation systems' versions whose behavior and performance results were compared by Dedicated Systems after testing them with its evaluation testing suite on the same x86 platform (Intel Pentium MMX).



```
done.
Activating swapfile swap...done.
Setting up networking....
Configuring network interfaces...Internet Systems Consortium DHCP Client
 V3.0.4
Copyright 2004-2006 Internet Systems Consortium.
All rights reserved.
For info, please visit http://www.isc.org/sw/dhcp/

 -> gelic_net_open:1386
Listening on LPF/eth0/00:19:c5:17:9d:6c
Sending on   LPF/eth0/00:19:c5:17:9d:6c
Sending on   Socket/fallback
DHCPDISCOVER on eth0 to 255.255.255.255 port 67 interval 5
DHCPOFFER from 192.168.88.129
DHCPREQUEST on eth0 to 255.255.255.255 port 67
DHCPACK from 192.168.88.129
bound to 192.168.88.197 -- renewal in 1535 seconds.
done.
Starting portmap daemon....
Setting console screen modes and fonts.
INIT: Entering runlevel: 2
Starting system log daemon: syslogd.
Starting kernel log daemon: klogd.
Starting portmap daemon...Already running..
* Not starting internet superserver: no services enabled.
Starting OpenBSD Secure Shell server: sshd.
Starting NFS common utilities: statd.
Starting periodic command scheduler: crond.

Debian GNU/Linux 4.0 ps3 tty1

ps3 login:
```

OS version	QNX NEUTRINO RTOS v6.5.0	Vanilla Linux 2.6.33.7
Applied patches	Patch 2530, from QNX Software Systems Ltd.	Real-time patch v30.

Table 1: The evaluated OSs

For **QNX Neutrino 6.5**, Patch 2530 was applied. This patch introduces a fix to the io-pkt network stack where a timer pulse implementation is used instead of attaching a handler to the timer interrupt. This patch significantly improves clock tick processing times and results in improved real time performance.

For **"Vanilla" Linux 2.6.33.7**, real-time patch rt-30 was applied to provide some real time characteristics for the Linux kernel. This RT patch was the latest version officially released by OSADL.

2.2 Hardware

We conducted our tests on the same x86 platform. This platform has the following characteristics:

- Motherboard: Chaintech 5TTMT M201 with a 33MHz PCI bus
- BIOS: Award BIOS v4.51PG
- CPU: Intel Pentium 200MHz MMX Family 5 Model 4 Stepping 3 (with 32KB L1 Cache)
- RAM: 256 MB
- Network interface card: Realtek RTL8139C(L)
- VMETRO PCI exerciser in PCI slot 3 (PCI interrupt level D, local bus interrupt level 10)
- VMETRO PBT-315 PCI analyser in PCI slot 4.
- External and CPU internal cache was enabled during the tests.

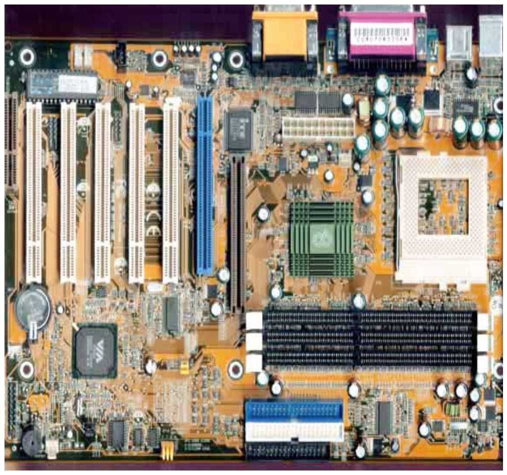


Figure 2: The hardware on which the tests were conducted

The framework 2.9 used for this report has the Pentium MMX 200 MHz as X86 reference platform. This processor has been used in a lot of X86 based systems some years ago. Although today no new designs use this processor, we continue to use it as reference in order to be capable to compare RTOS and also to compare with other (newer) platforms and see the enhancements in the field compared to 10 years ago.
This processor has only a limited cache and in this way the results are not that much influenced by the caching behaviour. As such, we are close to pure real-time behaviour. Cache is important for average performance enhancement. However it introduces a lot of uncertainty in the code execution with increased cache size. This report is about the worst case performance and we should exclude as much as possible the cache influence.
Also, the use of a slow processor will reveal more easily some behaviour aspects of the OS where otherwise these fine-grained differences would not be measurable.

3 Evaluation results overview

This section presents the overall ratings and evaluations based on key tests.

3.1 *Dedicated Systems' ratings for the tested RTOSs*

Here are Dedicated Systems' overall ratings for the tested OSs after testing them:

QNX SOFTWARE SYSTEMS QNX Neutrino 6.5.0	 Linux 2.6.33.7-rt30
★★★★★	★★★

Table 2: Overall ratings for the evaluated OSs

3.2 *Rating Criteria*

After testing each OS using the Dedicated Systems Evaluation Testing Suite, we used a star system to give each OS a rating based on the performance and behavior results. The maximum number of stars that an OS can receive is five (5).

Rating	Availability of real-time requirements	Performance, behavior and interrupts testing results
★★★★★	The OS works correctly out-of-the-box. No fear about the kernel configuration	Excellent (Hard RT is met)
★★★★	The OS works correctly out-of-the-box. No fear about the kernel configuration	Very good (Hard RT is met)
★★★	Special attention and knowledge are required to correctly configure the kernel	Good (only soft RT is met)
★★	Special attention and deep knowledge are required to correctly configure the kernel	Bad (even soft RT problems)
★	Correctly configuring the kernel is problematic	Problematic
☹	NO RT capabilities	Fails (for hard and soft RT applications)

Table 3: criteria used to evaluate OSs

3.3 Positive and negative points for each OS

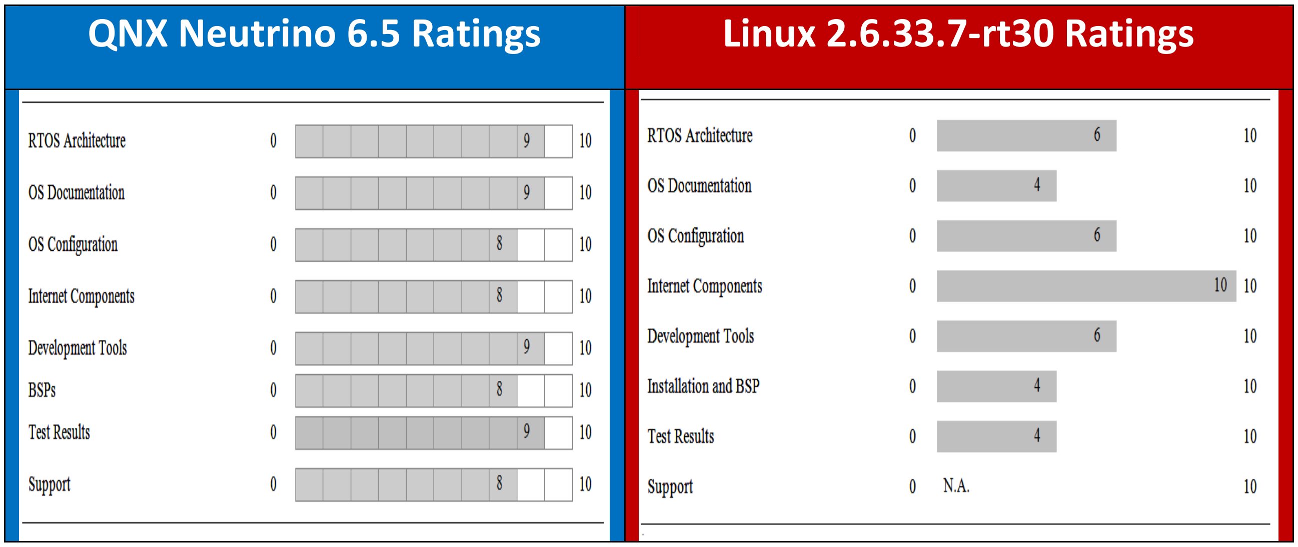
	QNX SOFTWARE SYSTEMS QNX Neutrino 6.5.0	 Linux 2.6.33.7-rt30
Positive Points 🙂	- Excellent architecture for a robust and distributed system. - Very fast and predictable performance. - Large number of board support packages (BSP) and drivers can be easily downloaded. - The availability of documentation is considered to be more than the average. - Efficient and user friendly (IDE)	- No license fees - Source code available - Extensible
Negative points 🙁	- Not all code is available in source code. Customers can apply for source access.	- The real-time characteristics are present only when everything is configured and built correctly. - GPL is not completely free! - Setting up a complete embedded target from scratch is a daunting task.

Table 4: positive and negative points found in each evaluated OS

3.4 Ratings by category

The table below presents the "ratings by category" comparison for the OSs evaluated. For a detailed description of the rating criteria, see [Doc. 2].

For more details about the OS, please see the relevant theoretical evaluation reports: QNX Neutrino 6.5 in [Doc 4] and RT Linux 2.6.33.7 in [Doc 5].



QNX Neutrino 6.5 Ratings	Min	Rating	Max	Linux 2.6.33.7-rt30 Ratings	Min	Rating	Max
RTOS Architecture	0	9	10	RTOS Architecture	0	6	10
OS Documentation	0	9	10	OS Documentation	0	4	10
OS Configuration	0	8	10	OS Configuration	0	6	10
Internet Components	0	8	10	Internet Components	0	10	10
Development Tools	0	9	10	Development Tools	0	6	10
BSPs	0	8	10	Installation and BSP	0	4	10
Test Results	0	9	10	Test Results	0	4	10
Support	0	8	10	Support	0	N.A.	10

Table 5: Ratings of the evaluated OSs

QNX Neutrino stands out as a clearly superior real-time OS compared to the other OSs evaluated. In addition to its design, which is much more robust and very easy to debug, even at the driver level, the data from our tests for this OS confirm that its real-time behaviour is considerably better than that of the other OSs. Further, this OS it is very well documented, and users do not have to worry about kernel configuration, as the OS kernel is always configured correctly.

The chief advantage of Linux is its open source licensing (no run-time fees). Note, however, that the GPL is not completely free, and investment is required to build a marketable system. For instance, though demo systems can be built quickly with Linux, the debugging, tuning and verification required to build a stable system ready for long-term use is much more difficult. Projects using Linux OSs tend to require large development teams. Further, projects that brew their own Linux flavor will need kernel experts who understand, for a start, how to set the kernel configurations (both at build and at run-time) to obtain real-time behavior.

3.5 Tests Summary

This section presents a brief comparative summary of the most important evaluation tests performed on the OSs we tested.

Detailed comparisons can be found in the next chapter. More detailed information about each test and its importance can be found in the corresponding documents: QNX Neutrino 6.5 [Doc 6] and Linux 2.6.33.7-rt30 [Doc 5].

Note that in the comparison figures and tables:

- The lower values means better quality
- Values in the charts are in microseconds (µs)

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

The "clock tick processing duration" test examines the clock tick processing duration in the kernel. The clock tick processing time is important because it impacts latencies everywhere in the system. The test results are extremely important because the clock interrupt will affect all the other measurements performed.

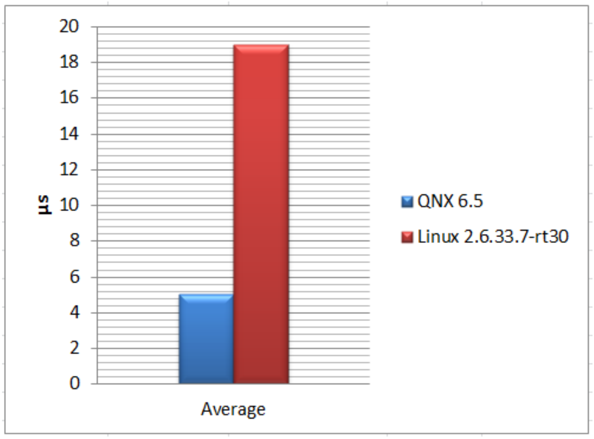


Figure 3a: Average clock interrupt duration

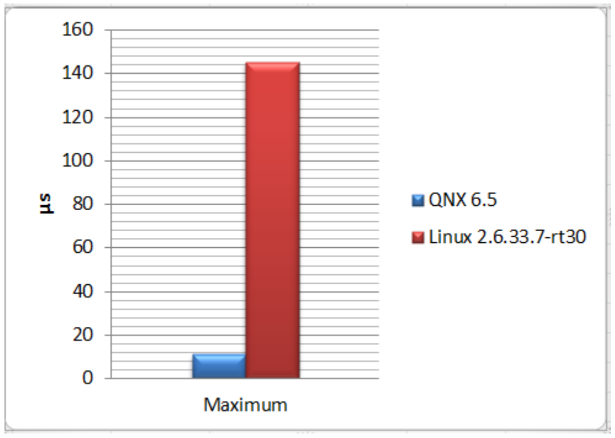


Figure 3b: Maximum clock interrupt duration

The clock tick processing time is important as it will impact latencies everywhere in the system. Since we are interested in real-time behavior, the maximum values are more important than the average values. Here you clearly see that QNX is designed from the ground up to keep real-time performance. In its worst case behavior, QNX is better than Linux by a factor of 13!

This is an important test because any latency in this test will show up in all other tests as the clock tick has typically the highest priority interrupt in a system.

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

The “latency between threads of same priority” test measures the time to switch between threads of the same priority using SCHED_FIFO policy. This test was performed four times, and each time using an increasing number of threads (2, 10, 128, and 1000) in order to generate the worst case behaviour.
The figures below present the thread switch latency with 1000 active threads in order to show the time values in the worst case. Data for evaluations with fewer threads are presented in the next chapter.

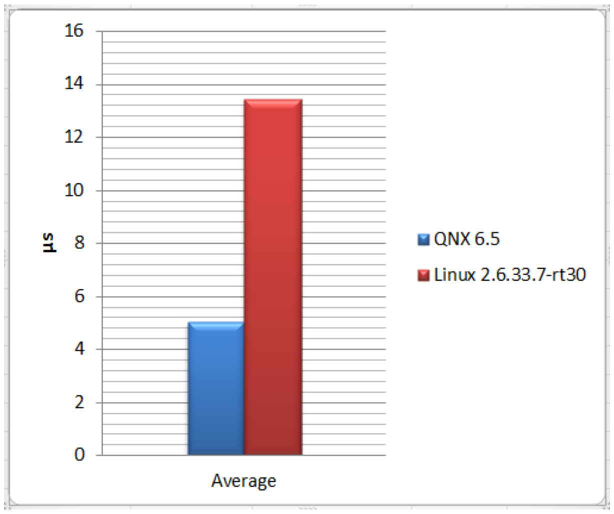


Figure 4a: Average latency between 1000 threads

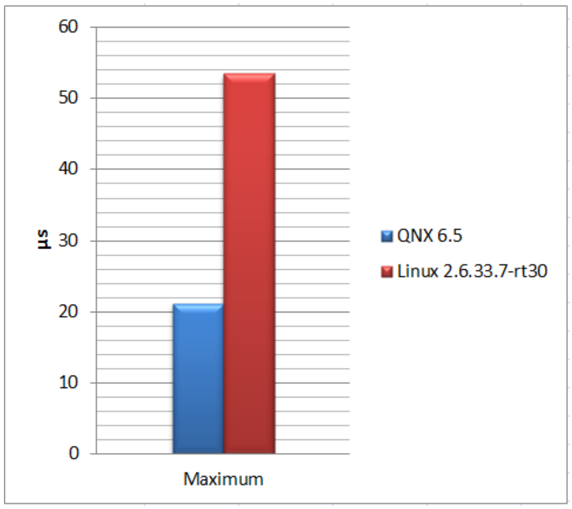


Figure 4b: Maximum latency between 1000 threads

Differences are smaller here; in this test, QNX Neutrino 6.5 outperforms RT Linux 2.6.33.7 for average latency; for maximum latency, QNX is better than Linux RT by a factor of 2.5.

3.5.3 Maximum sustained interrupt frequency (IRQ_S_SUS)

The "maximum sustained interrupt frequency" test measures the probability that an interrupt might be missed. In other words, it attempts to answer the question: Is the interrupt handling duration stable and predictable?

In this test, 1 billion interrupts are generated at specific interval rates. Our test suite measures whether the system under test misses any of the generated interrupts. The test is repeated with smaller and smaller intervals until the system under test is deemed to no longer handle the interrupt load. RT Linux functioned properly as long as interrupt levels were 150us or greater while QNX 6.5 was successful in servicing interrupts generated every 25 µsec. Remark that this test measures the worst case of the best case: due to the short time between interrupts, the interrupt handler tends to be cached.

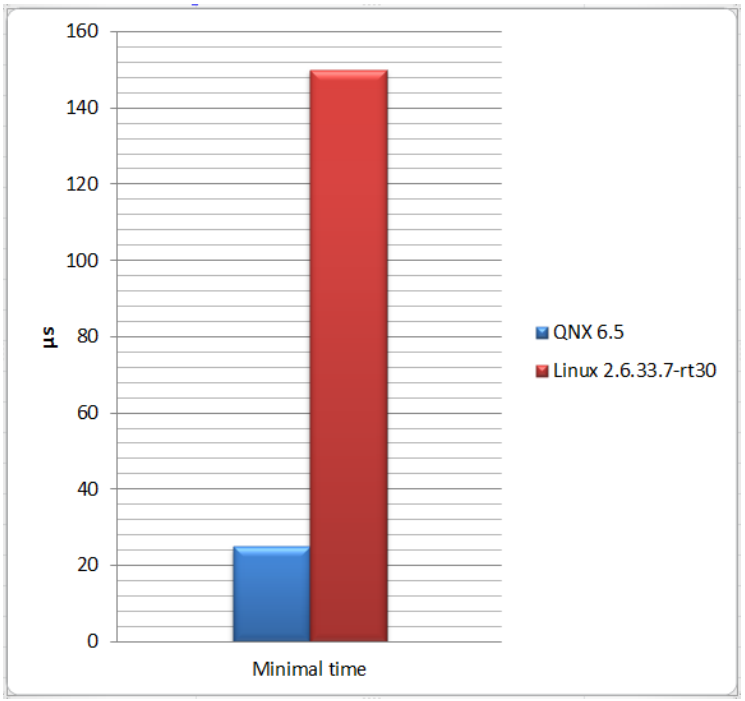


Figure 5: The minimal interrupt period required in order not to lose any of the 1 billion interrupts

QNX Neutrino fared best in handling the interrupts by successfully servicing interrupts generated every 25µs while Linux RT functioned properly as long as interrupt levels were 150µs or greater.

The long clock tick duration in Linux clearly has its impact which is seen in this test as QNX is better than Linux by a factor of 6!

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

The "mutex acquire-release timings in the contention case" test measures the time needed to acquire and release a mutex using priority inheritance. The acquire time is measured from the moment the higher priority thread requests the mutex until the moment the lower priority thread owning the mutex activates. The release time is measured from the moment the lower priority thread releases the mutex until the moment the higher priority thread is activated. As a result the total time spent on a locked mutex is thus the sum of the acquisition time + release time + the time the lock is taken by the lower priority thread.

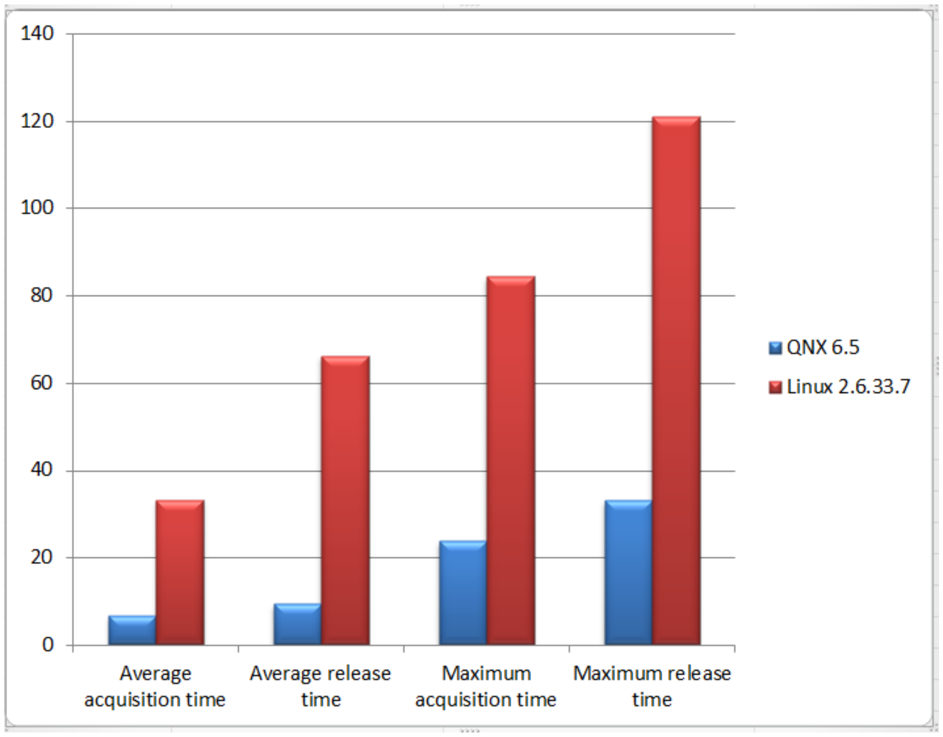


Figure 6: Mutex <u>average</u> and maximum acquire-release time in the <u>contention case</u>

Again QNX is better by factors. In the worst case, it is better by a factor of 4. On average, the difference is an even factor of 6.

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

The “mutex acquire-release timings: no-contention case” test measures the overhead incurred using a lock when a thread is not locked by another thread.

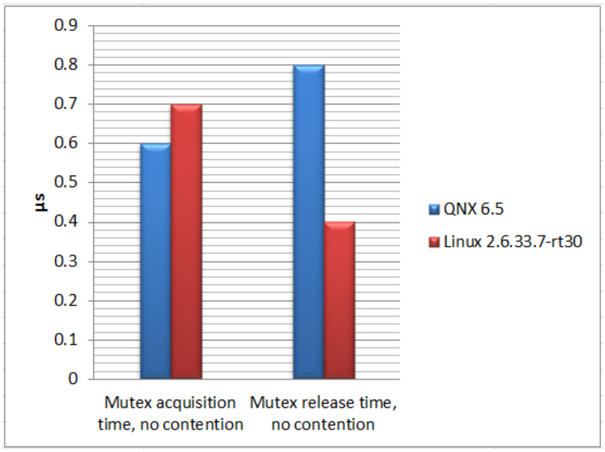


Figure 7a: Mutex <u>average</u> acquire-release time: no-contention case

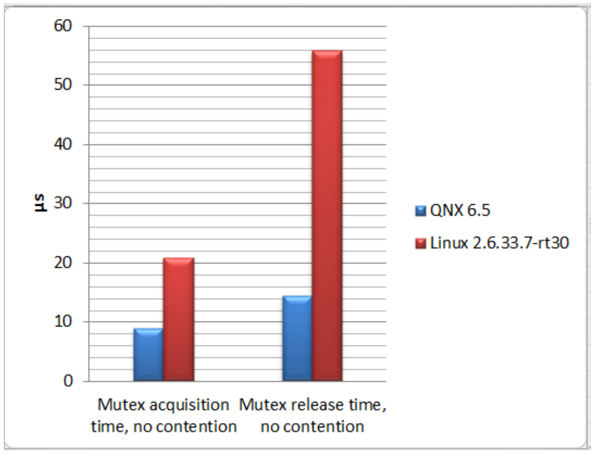


Figure 7b: Mutex <u>maximum</u> acquire-release time: no-contention case

The average results for this test are effectively equal. Note that this operation can be affected by clock tick duration time. As a result, maximum release times are much greater with RT Linux and QNX wins by a large margin.

4 Comparison Details

This section presents the detailed test results and the comparison between the evaluated OSs.

4.1 Clock tests (CLK)

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

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

The "clock tick processing duration" test examines the clock tick processing duration in the OS kernel. The test results are extremely important, as the clock interrupt will affect all the other performed measurements. The table below shows the average and maximum clock interrupt duration for the two tested OSs.

Clock interrupt duration	Average	Max
QNX Neutrino 6.5	5 µs	11 µs
Linux 2.6.33.7-rt30	19 µs	145 µs

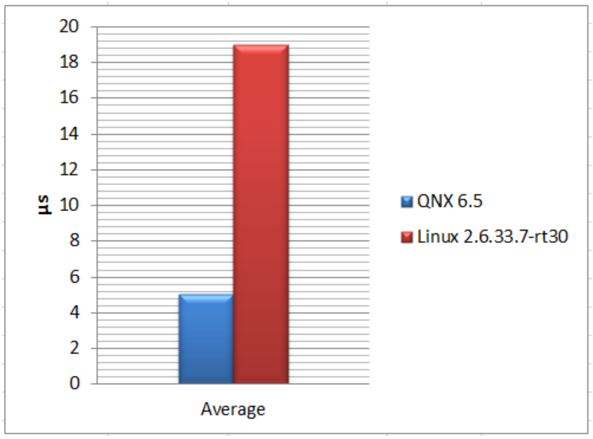


Figure 8a: Average clock interrupt duration

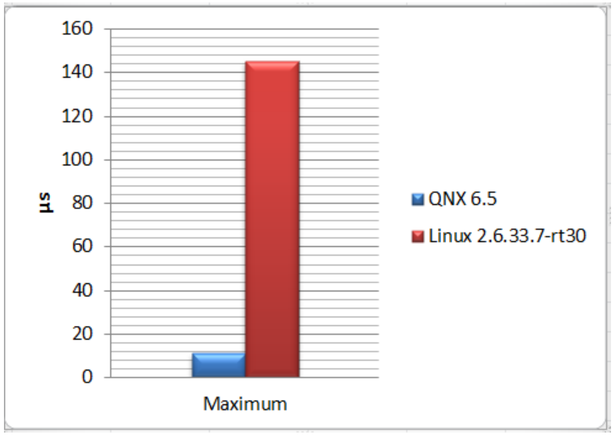


Figure 8b: Maximum clock interrupt duration

The clock tick processing time is important because it impacts latencies everywhere in the system. And as we are interested in real-time behavior, the measurements for maximum processing times are more important than the measurements for average processing times. Our testing showed that the traditional RTOS QNX Neutrino performs much better than Linux.

4.2 Thread tests (THR)

"Thread tests" measure the scheduler performance.

4.2.1 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?

The following scenarios were checked in the test:

- If a thread is created with a lower priority than the creating thread, can we be sure that it will not be activated until the creating thread is finished?
- If a thread is created with the same priority as the creating thread, is it placed at the end of the ready queue?
- When yielding after it was created by a thread of the same priority (as in the previous scenario), does the newly created thread become active?
- If a thread is created with a higher priority than the creating thread, does this new thread become activated immediately?

QNX SOFTWARE SYSTEMS QNX Neutrino 6.5.0	 Linux 2.6.33.7-rt30
Successfully passed this test	**Successfully passed this test**

Table 6: Results for the thread creation test

QNX Neutrino 6.5 and Linux RT passed this test successfully without any problems.

4.2.2 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)!

QNX SOFTWARE SYSTEMS QNX Neutrino 6.5.0	Linux 2.6.33.7-rt30
Successfully passed this test	passed this test ⚠

Table 7: Results of the round-robin test

⚠ Note that:

- For the **Linux** scheduler, the initial time slice of a created thread is 10 times greater than other slices (1second instead of the default 100milliseconds (ms)).

4.2.3 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.

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Thread switch latency, 2 threads	2.4 µs	17.2 µs	5.3 µs	53.8 µs
Thread switch latency, 10 threads	3.0 µs	15.6 µs	7.3 µs	54.4 µs
Thread switch latency, 128 threads	5.1 µs	32.2 µs	11.4 µs	49.0 µs
Thread switch latency, 1000 threads	5.0 µs	21.1 µs	13.4 µs	53.4 µs

Table 8: Thread switch latency between x threads, in µs

On QNX, we see that the average performance at 1000 threads is 0.1 µs better than at 128 threads. Let's say they are very close. 0.1µs is about measurement accuracy, and if you have bad luck with a couple of clock ticks tacking a bit longer, then it looks like it is better. We are pretty sure that if we run this test100 times, both results will be close and probably a bit longer for 1000 threads....

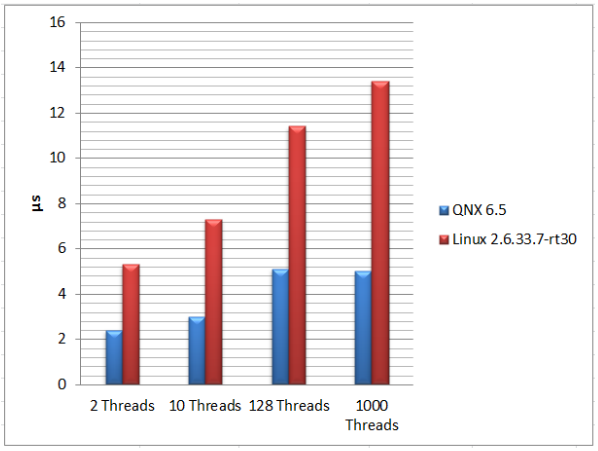


Figure 9a: Average switch latency between x threads

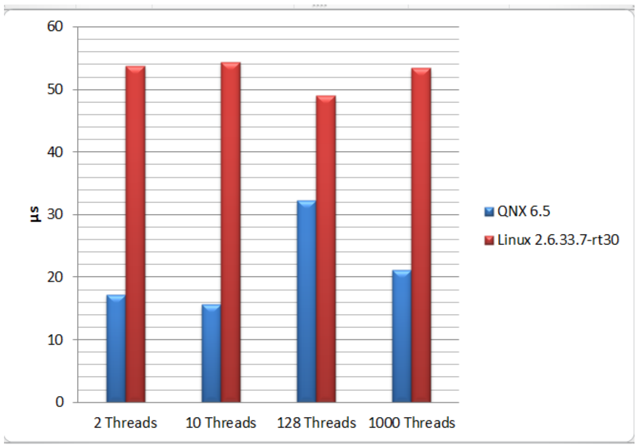


Figure 9b: Maximum switch latency between x threads

The impact of the caches on the average results is clearly observable (Figure 9a): the more threads there are to switch between, the more there are caches misses. The maximum values (Figure 9b) depend largely on the clock tick duration.

4.2.4 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.

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Thread creation, never run	215 µs	248 µs	213.4 µs	621.µs
Thread deletion, never run	152 µs	294 µs	371.7 µs	5044 µs
Thread creation, run and terminate	217 µs	245 µs	339.7 µs	738.5 µs
Thread deletion, run and terminate	15.5 µs	53.0 µs	13.7 µs	60.1 µs
Thread creation, run and block	214 µs	248 µs	347.1 µs	736.6 µs
Thread deletion, run and block	155 µs	295 µs	211.8 µs	268.7 µs

Table 9: Thread creation and deletion in different scenarios, in µs

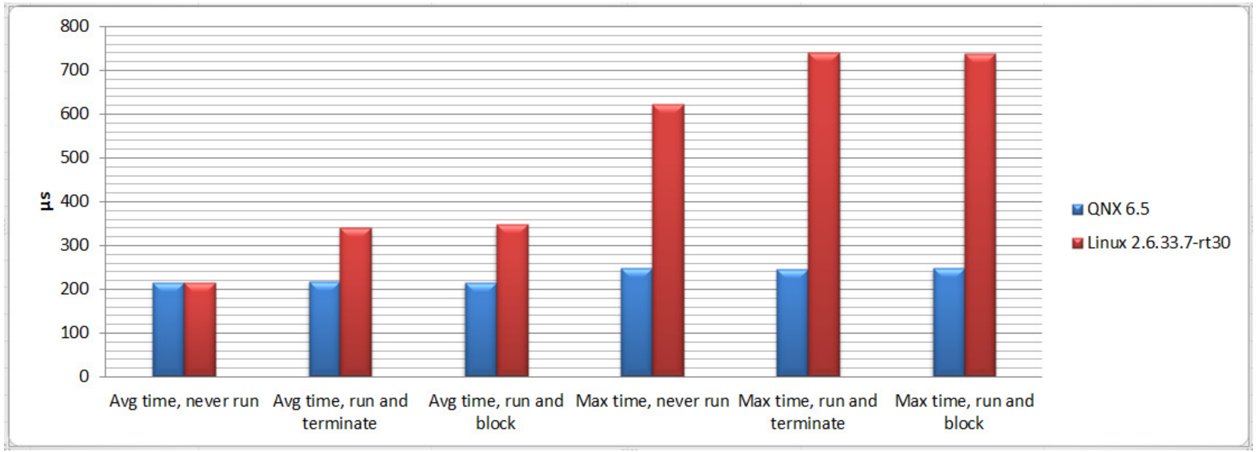


Figure 10a: Average and Maximum times for thread creation

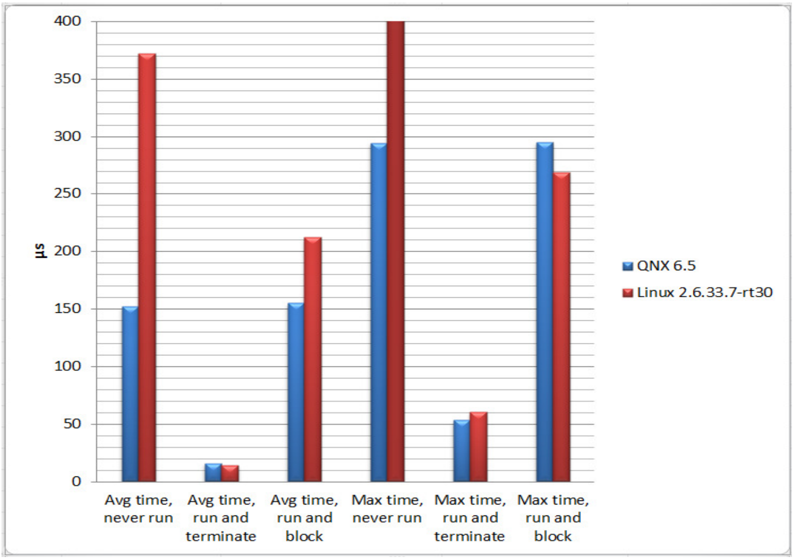


Figure 10b: Average and Maximum time for thread deletion

Note: the value of (Max time, never run) in figure 10b is very high (5044 µs) and it is not completely shown in the mentioned figure inorder to clearly observe the difference between the small values of the other parameters.

4.3 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.

With all the operating systems we tested, we did not specify a name to the semaphore when we conduct our tests. An unnamed semaphore cannot be used between processes. This limitation does not necessarily mean that the implementation with an unnamed semaphore does not use round-trips to the kernel.

4.3.1 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.

QNX SOFTWARE SYSTEMS QNX Neutrino 6.5.0	Linux 2.6.33.7-rt30
The semaphore behaves correctly as a protection mechanism	**The semaphore behaves correctly as a protection mechanism**

Table 10: The results of the semaphore locking mechanism test

4.3.2 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.

QNX SOFTWARE SYSTEMS QNX Neutrino 6.5.0	 Linux 2.6.33.7-rt30
Successfully passed this test	**Successfully passed this test**

Table 11: The results of the semaphore releasing mechanism test

4.3.3 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.

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Semaphore creation time, used	3.8 µs	39.2 µs	<0.1 µs	19.3 µs
Semaphore deletion time, used	3.6 µs	19.7 µs	<0.1 µs	<0.1 µs
Semaphore creation time, never used	3.7 µs	40.5 µs	<0.1 µs	57.2 µs
Semaphore deletion time, never used	3.3 µs	21.1 µs	<0.1 µs	0.3 µs

Table 12: The results of semaphore creation and deletion in different scenarios, in µs

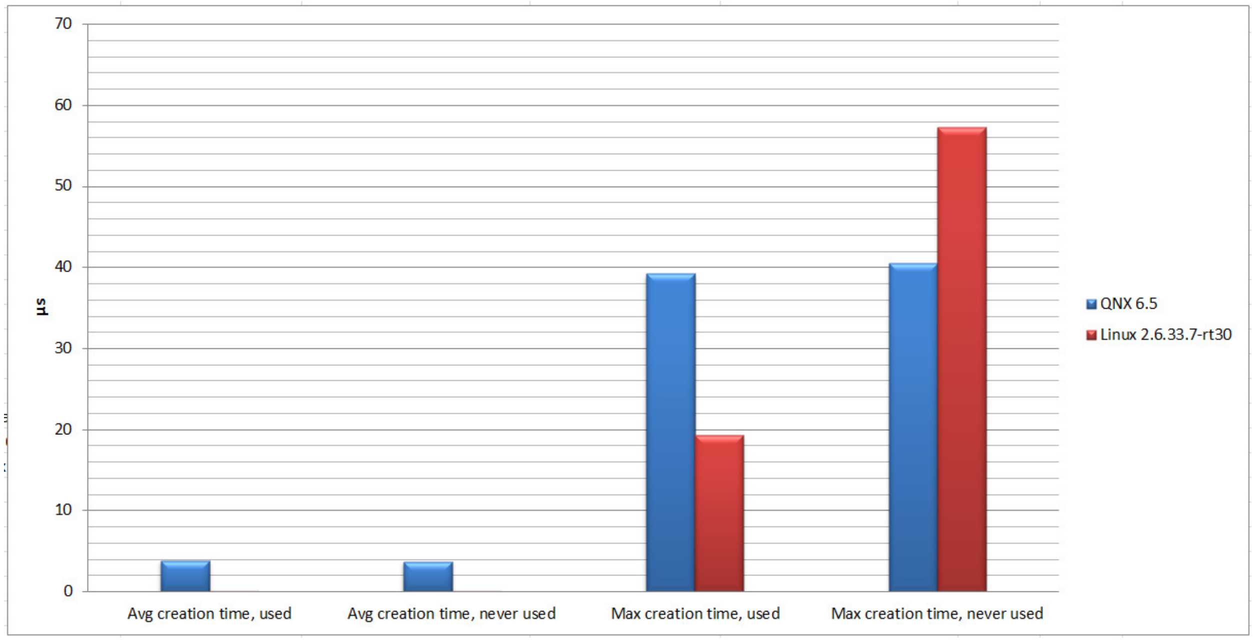


Figure 11a: Average and Maximum time for creating a Semaphore

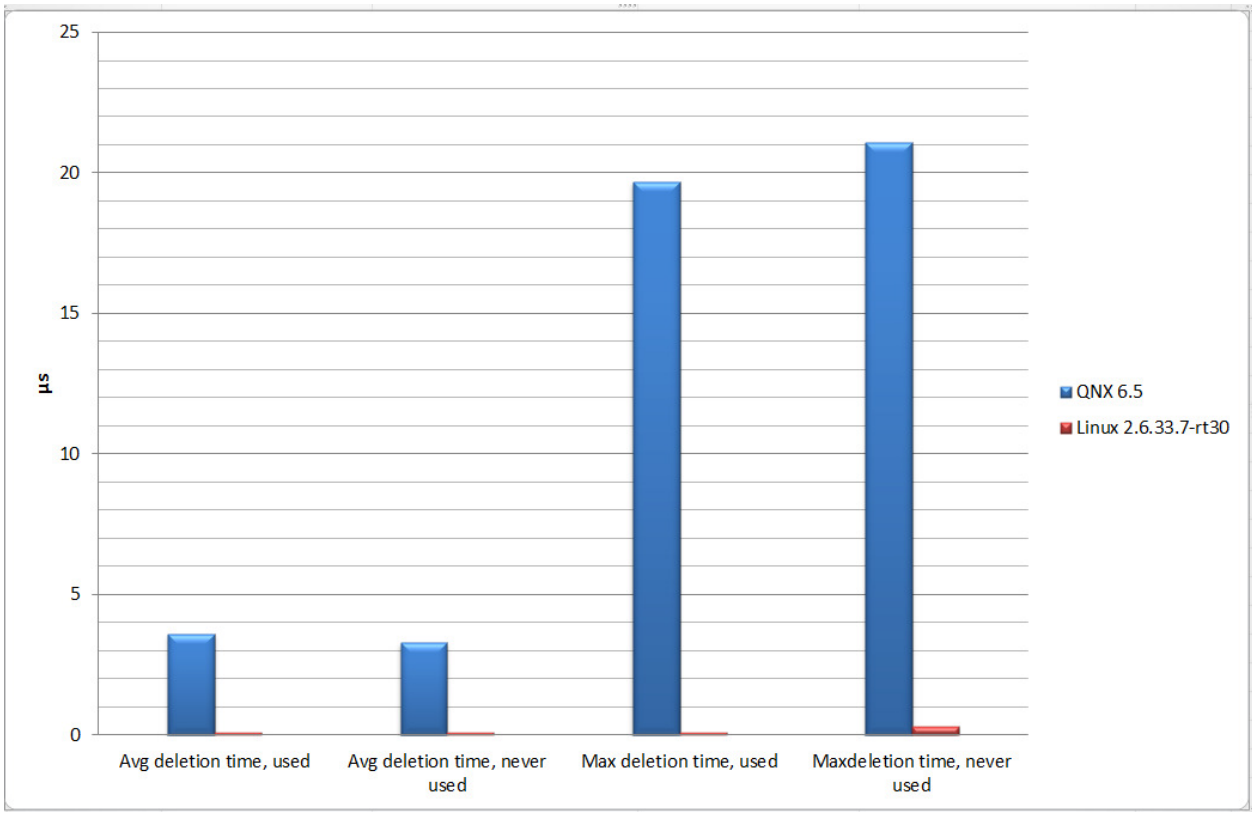


Figure 11b: Average and Maximum time for deleting a Semaphore

4.3.4 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.

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Semaphore acquisition time, no contention	2.5 µs	17.9 µs	<0.1 µs	58.0 µs
Semaphore release time, no contention	2.4 µs	14.2 µs	<0.1 µs	26.1 µs

Table 13: The semaphore acquisition and release times in contention case, in µs

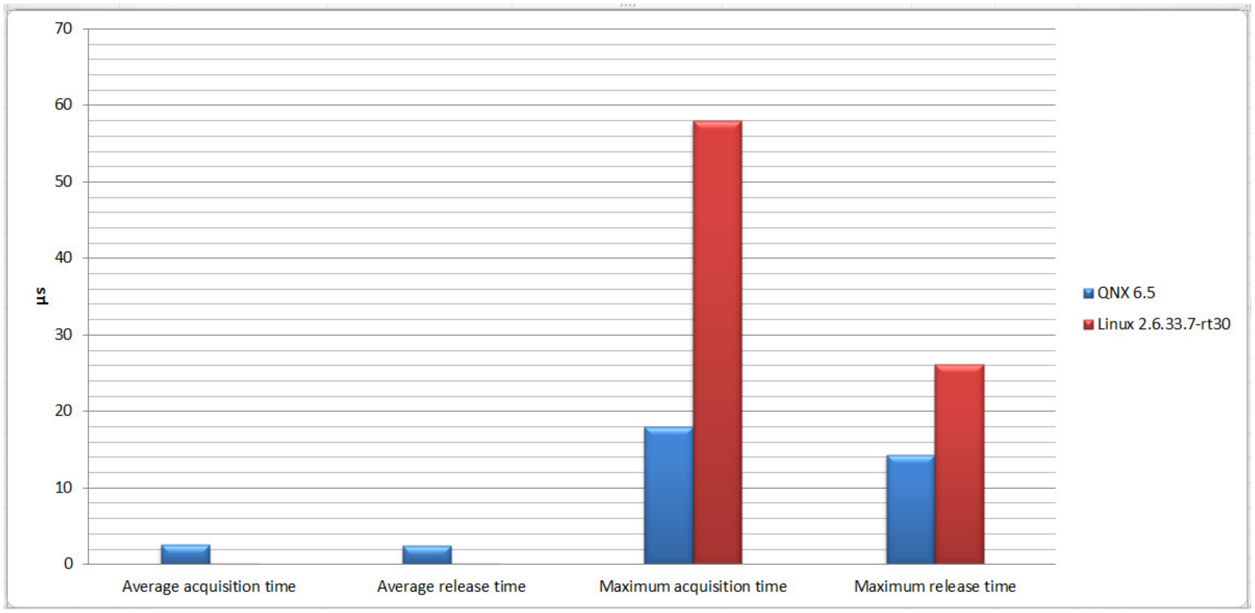


Figure 12: Semaphore acquire-release time: <u>no contention</u>

4.3.5 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. For more precise understanding of our test, please see the expanded diagrams showing a small time frame (e.g. one test loop). These diagrams are found in [Doc 6] for QNX Neutrino 6.5 and in [Doc 5] for RT Linux.

We create 128 threads with different priorities. The creating thread has a lower priority than the threads being created.

- When the thread starts execution, it tries to acquire the semaphore; but as it is taken, the thread stops and the kernel switch back to the creating thread. The time from the acquisition 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 compared with most other tests.
- After the last thread is created and is blocked on the semaphore, the creating thread starts to release the semaphore and this is 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).

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Semaphore acquisition time, contention	12.7 µs	37.6 µs	24.01 µs	219.4 µs
Semaphore release time, contention	12.1 µs	138 µs	36.9 µs	301.2 µs

Table 14: The semaphore acquisition and release times in contention case

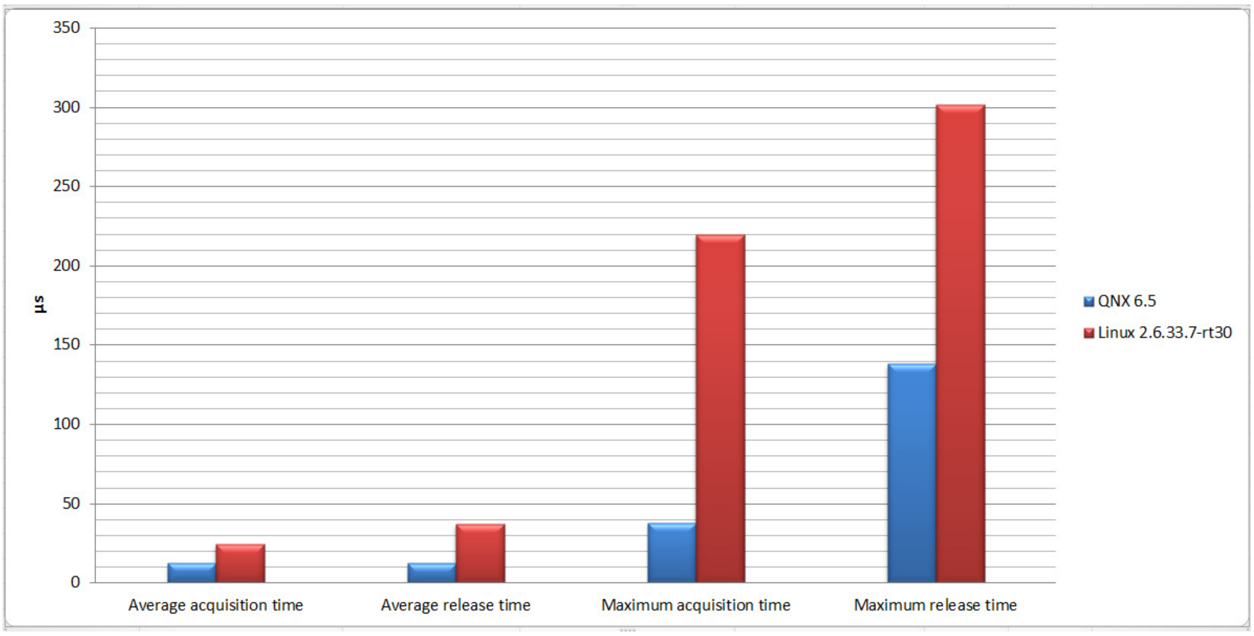


Figure 13: Semaphore acquire-release time: Contention case

4.4 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.

Our test will on purpose generate a priority inversion with three threads:

- Low priority thread having a lock
- Intermediate priority thread ready to run
- High priority thread running and requesting the lock owned by the low priority thread

If the mutex has some priority inversion avoidance mechanism present, the intermediate priority thread may not run until the lower priority thread released the mutex and the high priority thread finished its work.

Without such avoidance mechanism, the intermediate priority thread will start to run and thus delay the higher priority thread. Thus, as a result, priorities would be inverted!

4.4.1 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.

QNX SOFTWARE SYSTEMS QNX Neutrino 6.5.0	 Linux 2.6.33.7-rt30
Priority inversion behaves as expected	**Priority inversion behaves as expected**

Table 15: priority inversion avoidance mechanism test results

4.4.2 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 (MUT_B_ARC)" 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.

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Mutex acquisition time, contention	6.6 µs	23.7 µs	33.2 µs	84.3 µs
Mutex release time, contention	9.4 µs	33.1 µs	66.1 µs	120.9 µs

Table 16: Mutex acquire and release times, in contention case

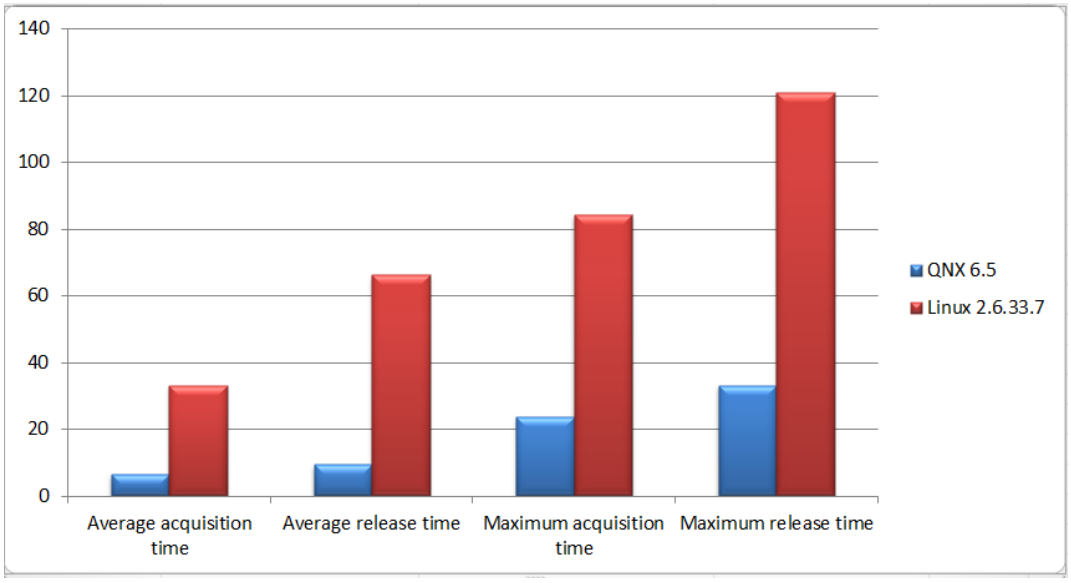


Figure 14: Mutex acquire-release time: contention

4.4.3 Mutex acquire-release timings: no-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.

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Mutex acquisition time, no contention	0.6 µs	8.8 µs	0.7 µs	20.8 µs
Mutex release time, no contention	0.8 µs	14.3 µs	0.4 µs	55.8 µs

Table 17: Results of the mutex acquire-release timing in no-contention case, in µs

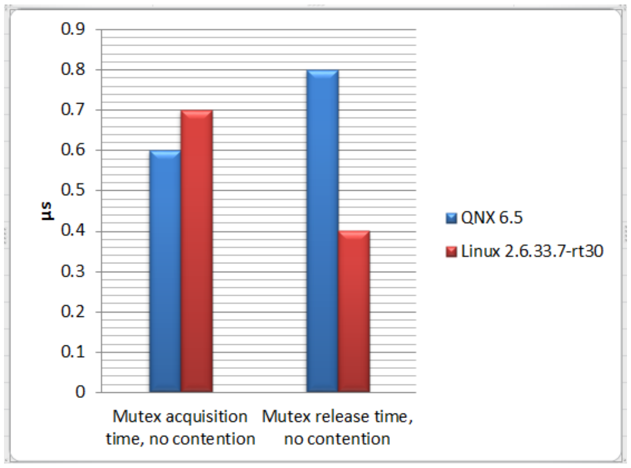


Figure 15a: Average mutex acquire-release time <u>no-contention</u>

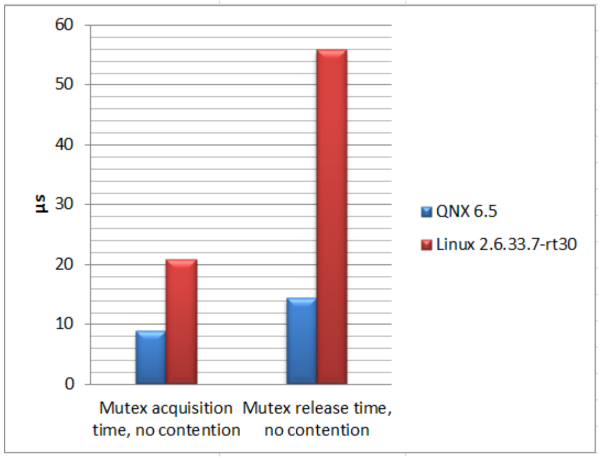


Figure 16b: Maximum mutex acquire-release time: <u>no-contention</u>

4.5 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,, 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 the independent interrupt source is not synchronised in any way with the platform under test.

4.5.1 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.

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Interrupt dispatch latency	1.8 µs	5.8 µs	8.5 µs	32.4 µs

Table 18: interrupt latency results in µs

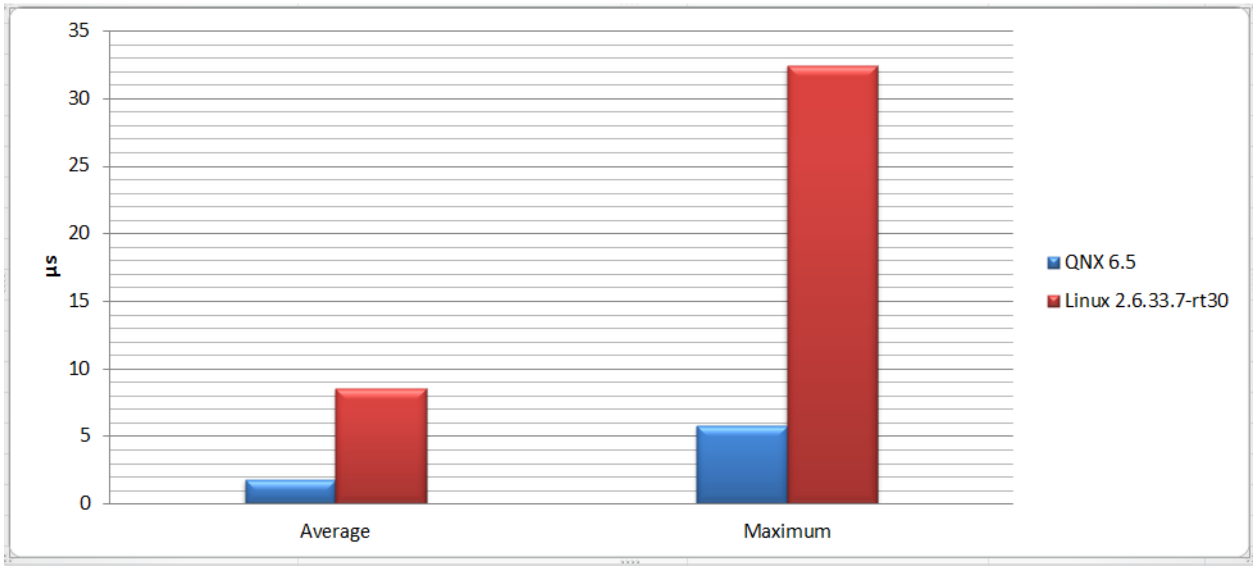


Figure 17: Interrupt dispatch latency

4.5.2 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.

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Dispatch latency from interrupt handler	1.4 µs	13.1 µs	2.7 µs	26.1 µs

Table 19: Interrupt dispatch latency in µs

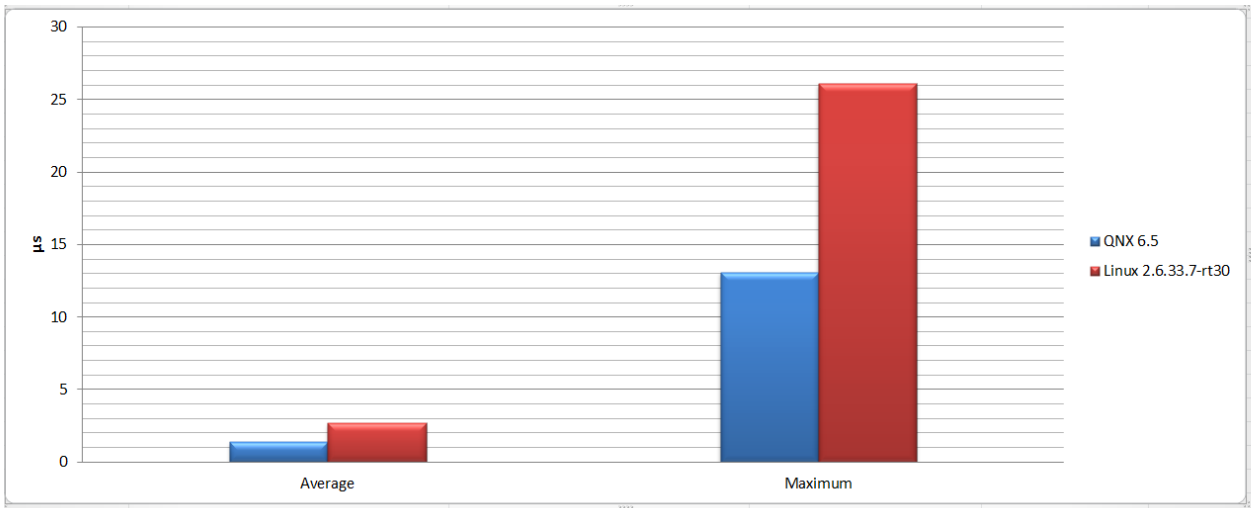


Figure 18: Dispatch latency from interrupt handler

4.5.3 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.

The OSs we evaluated do not all handle switching in the same way, and we tailored our tests to obtain comparable results:

- For QNX Neutrino, the interrupt handler emits an event to release a blocked thread. This blocked thread has the highest priority in the system.
- For Linux RT, a thread is blocked by using an `ioctl` call, and is released in the kernel module upon the interrupt.

This test measures the time the OS takes from the interrupt handler to the blocked thread (as a consequence this includes a thread switch).

Test	QNX		LINUX	
	Avg	Max	Avg	Max
Latency from ISR to waken-up thread	3.7 µs	15.0 µs	21.6 µs	76.4 µs

Table 20: Interrupt to thread latency, in µs

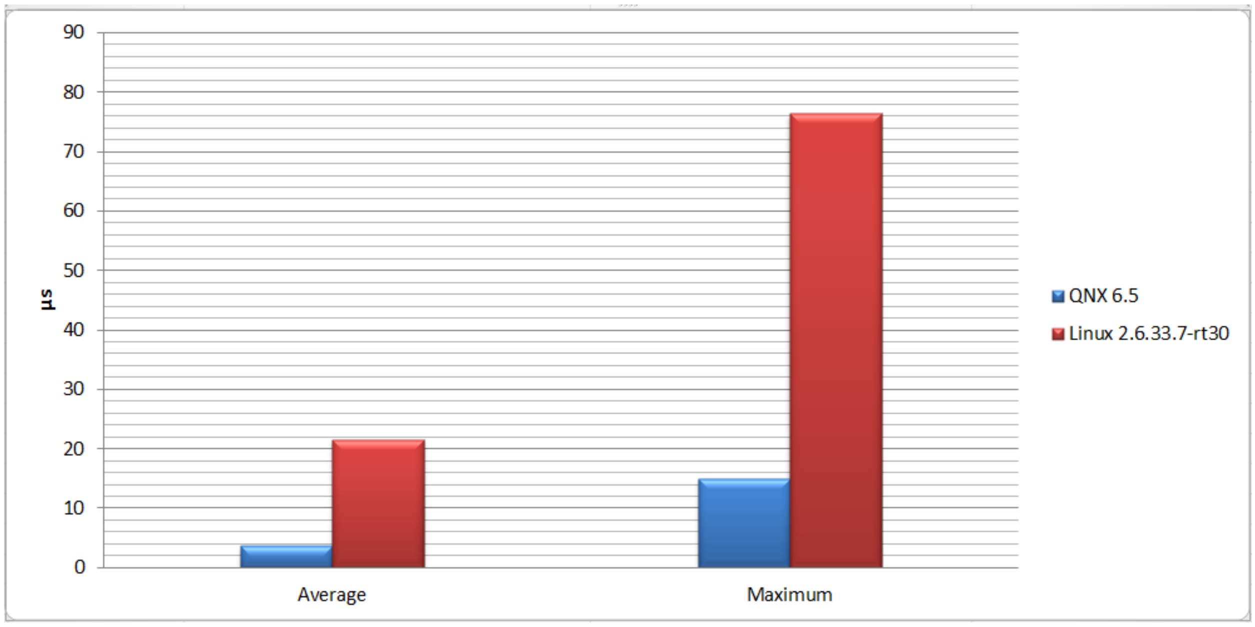


Figure 19: Latency from ISR to waken-up thread

4.5.4 *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?

In this test we load the system with a high load interrupt source which generates 100 million interrupts and determine at which interrupts frequency the OS begins to miss interrupts. The table below shows the minimum delay required between interrupts for the OSs tested to not lose any of the 100 million interrupts. Below this threshold, the OSs lost interrupts.
Note that this test presents the worst case of the best-case scenario: due to the high interrupt rate, the interrupt handler is expected to be in the cache all time.

This test shows clearly how good an OS can keep its real-time deadlines. Linux (with the RT_PREEMPT patch) needs 6 times more time not to lose any interrupt compared with QNX.

Test	QNX	LINUX
Minimal interrupt period required not to lose any of the 1000.000.000 generated interrupts.	25 µs	150 µs

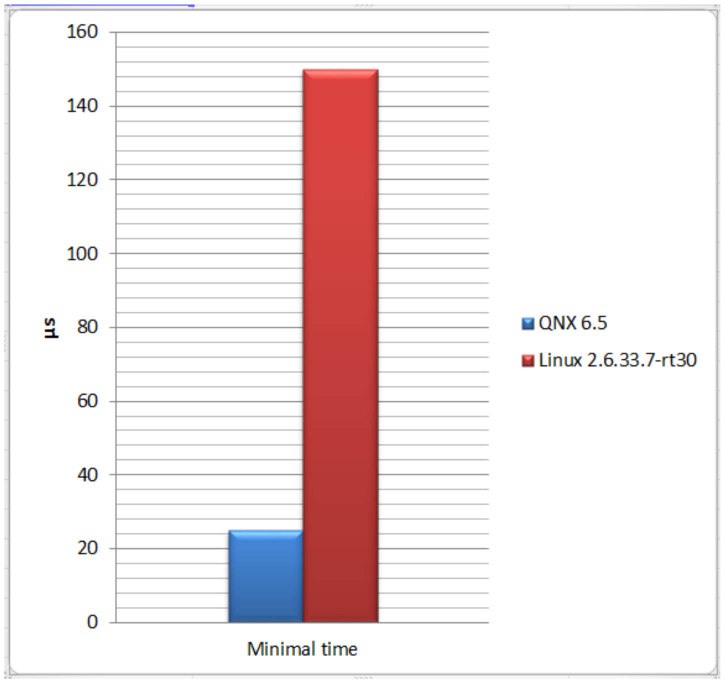


Figure 20: Minimal interrupt period required not to lose any of the 1000.000.000 generated interrupts.

5 Conclusion

First of all, both OSs can be qualified as RTOS.

Remark however that for Linux, you will need to take care of having a correct kernel configuration, both at build time as at run time. Further you will need to have correct libraries so the kernel features (like priority inheritance support on mutexes) can be used from application space.

QNX provides the real-time behaviour out-of-the-box, so there is nothing to worry about.

The results also clearly show that the real-time performance is much better for QNX. The worst case in the different test is typically multiple times larger on Linux compared with QNX. This can clearly be seen in the sustained interrupt test, where this is about a factor of 6.

6 Related documents

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

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

Doc. 2 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. 3 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. 4 QNX v6.5, Theoretical evaluation
This document presents the qualitative discussion of the OS
EVA-2.9-OS-QNX-65 Issue: 1 May 20, 2011

Doc.5 Linux technical evaluation report
This document presents the results of evaluating Linux on x86 platform (MMX)
EVA-2_9-TST-LINUXRT_2_6_33_7_2-rt30-x86 Issue: 1 May 30, 2011

Doc. 6 QNX v6.5, technical evaluation report
This document presents the results of evaluating QNX 6.5 on x86 platform (MMX)
EVA-2 9-TST-QNX-65-x86-01 Issue: 1 Sept 7, 2011

7 Appendix A: 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

DOCUMENT CHANGE LOG

Issue No.	Revised Issue Date	Para's / Pages Affected	Reason for Change
1.0	Sept 5, 2011	ALL	Initial report
1.1	Jan 3, 2012	ALL	Check-up the text and add some figures
1.2	Jan 5, 2012	All	Add figures
1.3	Jan 6, 2012	All	Modify figures
1.4	Jan 17, 2012	All	Insert the feedback
1.8	Jan 23, 2011	All	Review and addition of stars rating
2.0	March 5, 2012	All	Finalize it