Comparison of QNX Neutrino, Windows CE7, Linux RT and Android (RT) operating systems on ARM processor

Copyright

Disclaimer

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

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 authorisation.
6. **INDEMNIFICATION**. You agree to indemnify and hold harmless Dedicated Systems Experts against any damages or liability of any kind arising from any use of this product other than the permitted uses specified in this agreement.
7. **DISCLAIMER OF WARRANTY**. All documents published by Dedicated Systems Experts on the World Wide Web Server or by any other means are provided "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. This disclaimer of warranty constitutes an essential part of the agreement.
8. **LIMITATION OF LIABILITY**. Neither Dedicated Systems Experts nor any of its directors, employees, partners or agents shall, under any circumstances, be liable to any person for any special, incidental, indirect or consequential damages, including, without limitation, damages resulting from use of OR RELIANCE ON the INFORMATION presented, loss of profits or revenues or costs of replacement goods, even if informed in advance of the possibility of such damages.
9. **ACCURACY OF INFORMATION**. Every effort has been made to ensure the accuracy of the information presented herein. However Dedicated Systems Experts assumes no responsibility for the accuracy of the information. Product information is subject to change without notice. Changes, if any, will be incorporated in new editions of these publications. Dedicated Systems Experts may make improvements and/or changes in the products and/or the programs described in these publications at any time without notice. Mention of non-Dedicated Systems Experts products or services is for information purposes only and constitutes neither an endorsement nor a recommendation.
10. **JURISDICTION**. In case of any problems, the court of BRUSSELS-BELGIUM will have exclusive jurisdiction.

Agreed by downloading the document via the internet.

Contents

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 four real time operating systems (RTOSs). These OSs are:

- QNX Neutrino 6.5 patch 2530
- Windows Embedded Compact 7
- Linux 2.6.33.7.2-rt30
- Android Linux 3.0.1

The order in which we list the OSs is based on the overall results obtained by the OSs, with the OS with the best results listed first and the others following in descending order. This ordering is maintained throughout the whole report.

These RTOSs were evaluated on the same ARM platform (BeagleBoard-XM Rev C).

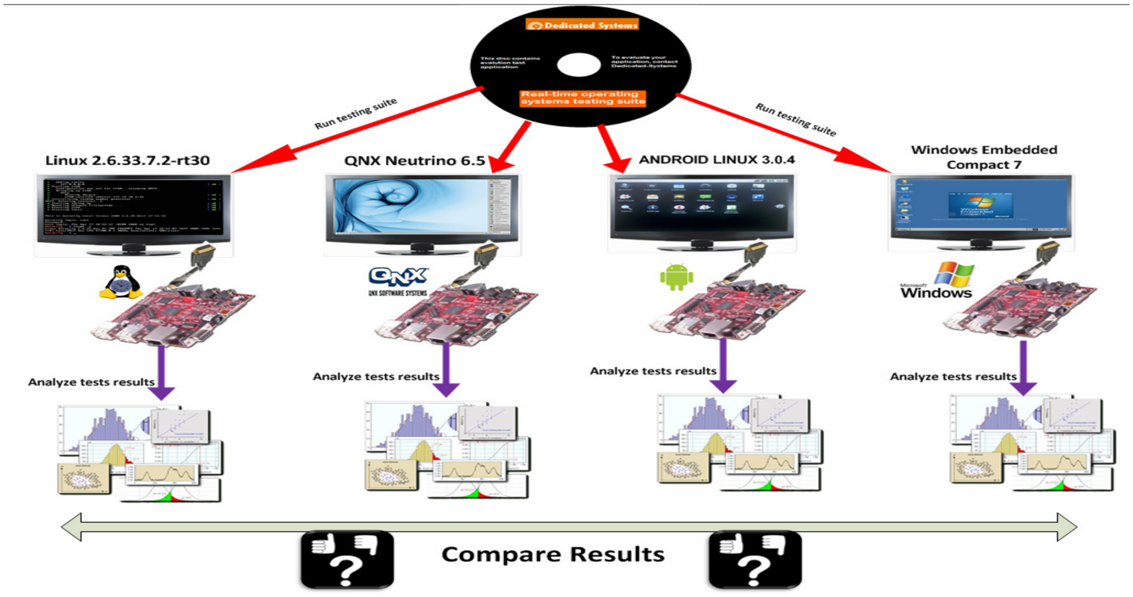


Figure 1: High level view of the evaluation procedure

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 ARM platform (BeagleBoard-XM Rev C).

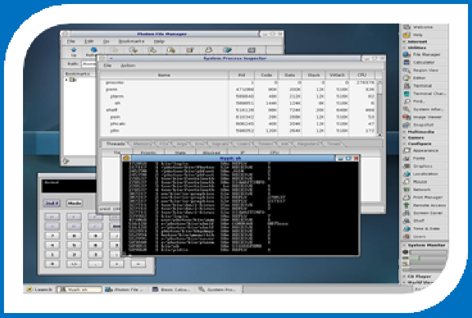
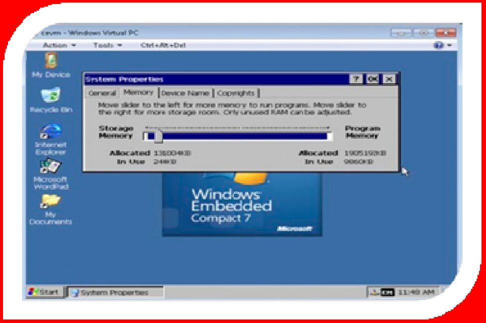
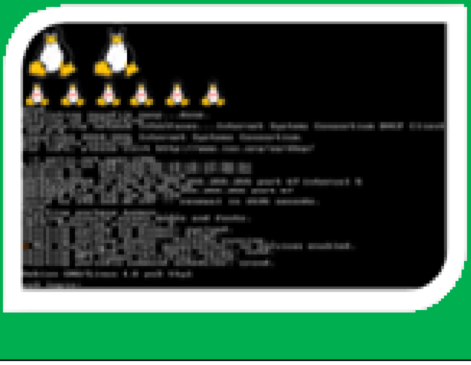
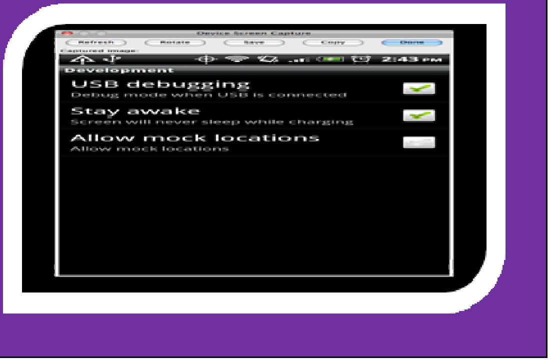
QNX Neutrino RTOS v6.5.0 with Patch 2530	Windows Embedded Compact 7
	 System Properties General / Memory / Device Name / Copyrights Move slider to the left for more memory to run programs. Move slider to the right for more storage room. Only unused RAM can be adjusted. Storage Memory — Program Memory Allocated 13100KB In Use 24KB — Allocated 190519KB In Use 9060KB
Vanilla Linux 2.6.33.7 with RT-30 Patch	**Android Linux 3.0.4**
	 Development USB debugging ☑ Debug mode when USB is connected Stay awake ☑ Screen will never sleep while charging Allow mock locations □ Allow mock locations

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 **Windows Embedded Compact 7**, no patches were applied.

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.

For **Android Linux 3.0.4**, no patches were applied

2.2 Hardware

We conducted our tests on the same ARM platform. This platform is a Beagle-XM Board Rev C with the 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

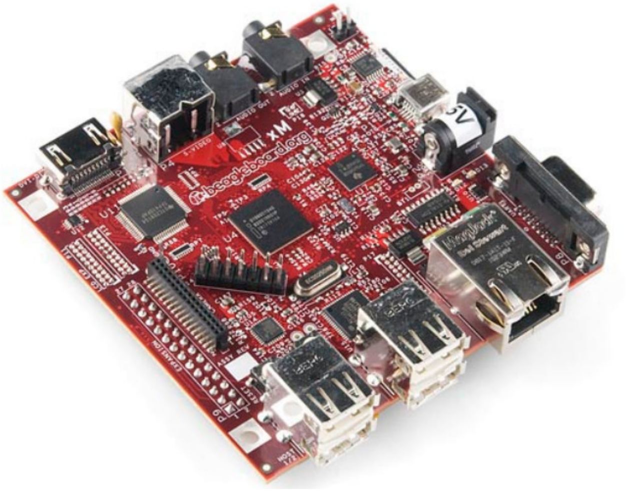


Figure 2: The ARM board on which the tests were conducted

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 Neutrino 6.5.0	Windows Embedded Compact 7	Linux 2.6.33.7-rt30	Android Linux 3.0.4
★★★★★	★★★★⯪	★★★	☹

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

Evaluated OS	Positive points ☺ ☺ ☺	Negative points ☹ ☹ ☹
QNX Neutrino 6.5.0	1) Excellent architecture for a robust and distributed system. 2) Very fast and predictable performance. 3) Large number of (BSPs) and drivers can be easily downloaded. 4) The availability of documentation and support is very high. 5) Efficient and user friendly (IDE)	1) Not all code is available in source code. Customers can apply for source access.
Windows Embedded Compact 7	1) All protection primitives use priority inheritance, which is a major plus for achieving real-time behavior. 2) Good debugging tools which are also available for kernel/driver debugging. 3) Very easy to install and to set-up a target (from templates). 4) provides the same flexibility as a 32-bit general purpose OS	1) A lot of background information is only available for CE6R3. 2) Customizing the kernel and adding custom drivers (BSP) stays a daunting task once you go away from the default configurations.
Linux 2.6.33.7-rt30	1) No license fees. 2) Source code available. 3) Extensible	1) The real-time characteristics are present only when everything is configured and built correctly. 2) GPL is not completely free… 3) Setting up a complete embedded target from scratch is a daunting task.
Android Linux 3.0.4	1) No license fees. 2) Source code available	1) No real-time characteristics at all! 2) Not meant to be used for any C/C++ applications. 3) Bionic C library, implemented semaphores and mutexes badly.

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], Windows Embedded Compact 7 in [Doc 8], Linux 2.6.33 in [Doc 6], and Android Linux 3.0.4 in [Doc 10].

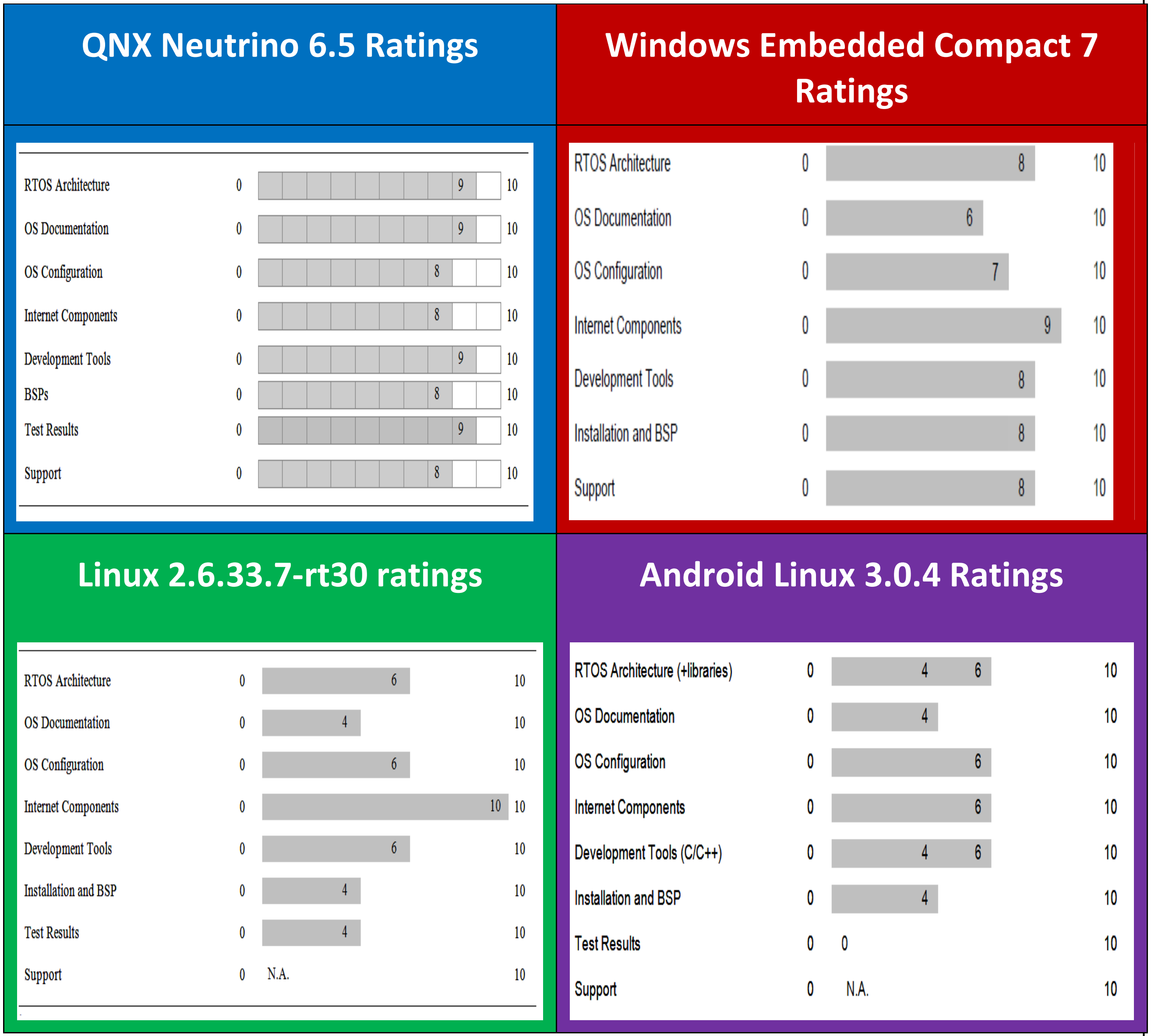


Table 5: Ratings of the evaluated OSs

3.4.1 QNX Neutrino 6.5 with Patch 2530

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.

3.4.2 Windows Embedded Compact 7

Like the QNX Neutrino RTOS, Microsoft Windows Compact Embedded 7 (CE7) is also a system built for real-time behavior. This OS has a large toolset, and many options are available for building different types of systems. However, this OS is a marginally slower than the QNX Neutrino 6.5, and the toolset for building drivers and develop BSPs is less intuitive than the ones supplied by QNX. Also like QNX Neutrino OS, it tested significantly better than Linux 2.6.33.7-rt30.

3.4.3 Linux 2.6.33.7 with rt30 Patch

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.

3.4.4 Android Linux 3.0.4

Android Linux 3.0.4 is different from the other evaluated OSs: it is not built for implementations requiring real-time characteristics, and Google makes no claim of real-time behavior. However, because there seems to be some interest in using this OS as a real-time OS, we evaluated it. We saw that not only the kernel but also the libraries are important to an OS's ability to meet real-time deadlines. We found that the Android Linux OS's failure to meet real-time deadlines was largely due to its libraries. The Android Linux Bionic library does not present all kernel features to the application; for instance, there is no mechanism in place to avoid priority inversion. Test results show that Android Linux cannot guarantee 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 5], Windows Compact Embedded 7 (CE7) [Doc 9], Linux 2.6.33.7-rt30 [Doc 7], and Android Linux 3.0.4 [Doc 10].

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.

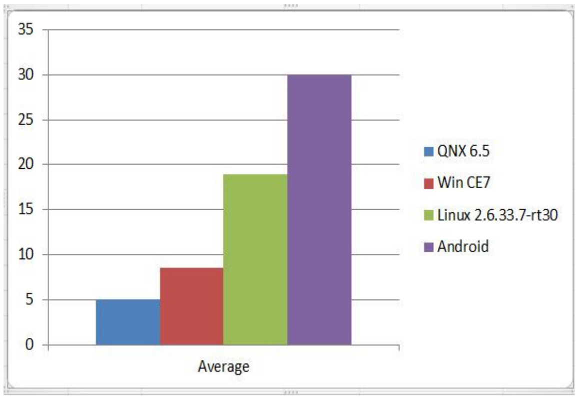


Figure 3a: Average clock interrupt duration

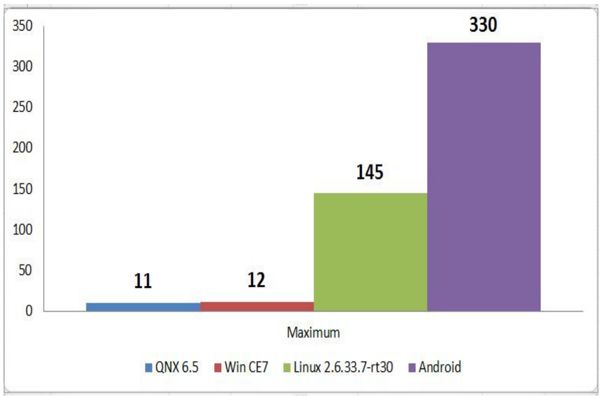


Figure 3b: Maximum clock interrupt duration

Since we are interested in real-time behavior, the maximum values are more important than the average values. From our results, it is clear that the traditional RTOSs (QNX and Windows CE) are still miles ahead of the Linux variants. The maximum clock durations (Figure 3b) for QNX Neutrino 6.5 and Windows CE7 is almost the same (11 for QNX Neutrino 6.5 and 12 for Windows CE7), while it is very high for the Linux 2.6.33.7-rt30 and Android Linux 3.0.4 OSs.

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.

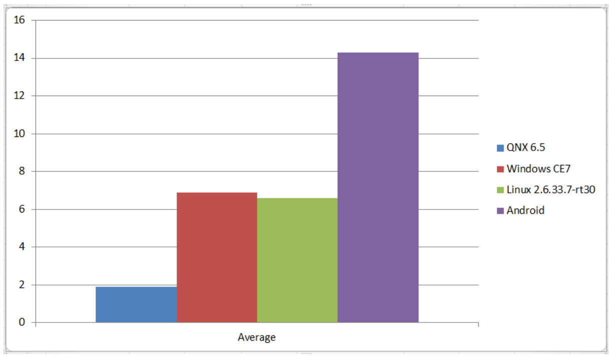


Figure 4a: Average latency between 1000 threads

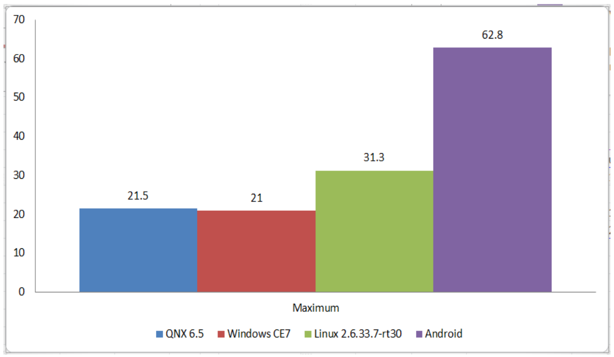


Figure 4b: Maximum latency between 1000 threads

In this test, QNX Neutrino outperforms Windows CE7 for average latency. Linux RT shows results similar to Windows CE7, while the average latency for Android Linux is more than twice that for Linux RT and Windows CE7, and more than seven times the latency for QNX Neutrino.

The maximum latency depends on the clock tick interrupts; so if the test would run long enough, we would see similar results as the clock tick duration test (All tests show this congruence, which is why the clock tick duration test is so important to evaluating an RTOS). For Linux RT, the maximum latency is somewhat greater than for QNX Neutrino and Windows CE7.

It is curious that for Android Linux (running kernel version 3.0.4) the thread switch latency is twice as long as with Linux RT (running kernel version 2.6.33.7).

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, 100 million 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.

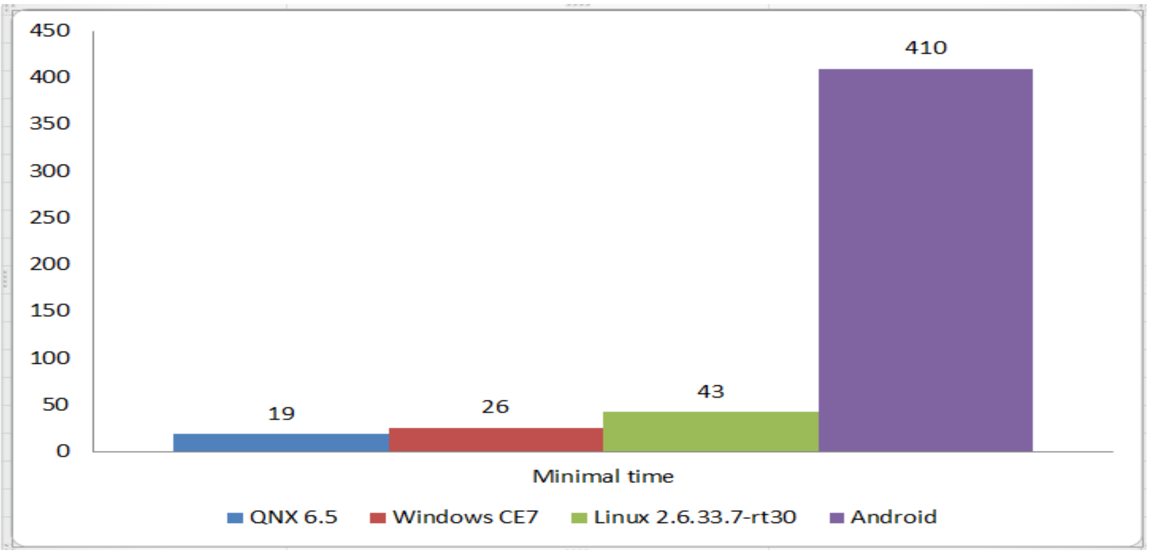


Figure 5: The minimal interrupt period required in order not to lose any of the 100 million interrupts

QNX Neutrino fared best in handling the interrupts by successfully servicing interrupts generated every 19µs. Windows CE7 was second best and was able to handle interrupts with a value of 26 µs, while Linux RT functioned properly as long as interrupt levels were 43µs or greater. Android Linux exhibited the poorest characteristics, and was not able to handle interrupts more frequent than 410 µs, which is about ten times slower than Linux RT and 20 times slower than QNX Neutrino.

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.

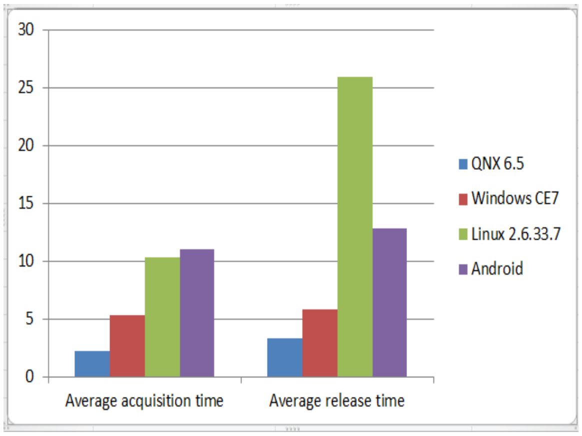


Figure 6a: Mutex <u>average</u> acquire-release time: <u>contention case</u>

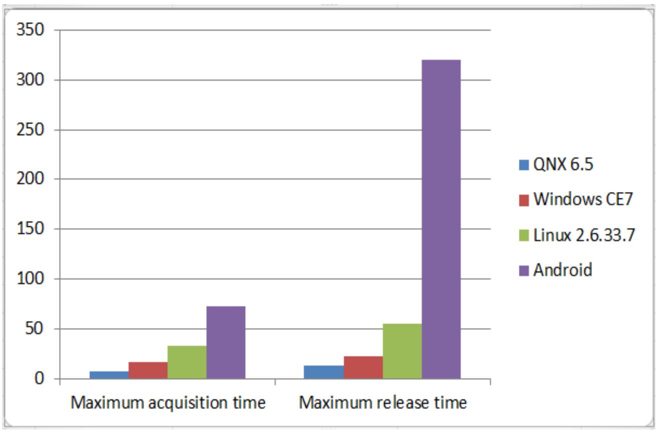


Figure 6b: Mutex <u>maximum</u> acquire-release time: <u>contention case</u>

The advantage of the classical RTOSs compared with Linux is clear again. We also noticed that the release time on Linux RT is longer than expected. This is probably caused by the priority inheritance mechanism which takes some overhead, but it is required for real-time behavior. As this inheritance mechanism is not used by Android, it has a better average release time, but of course a bad worst case delay. (the reason why Android average release time is much better than Linux RT).

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.

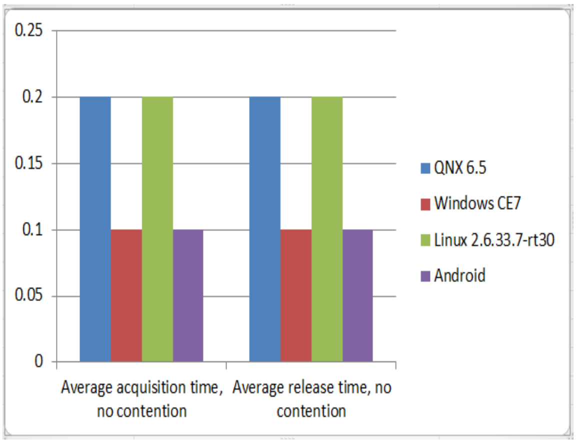


Figure 7a: Mutex <u>average</u> acquire-release time: no-contention case

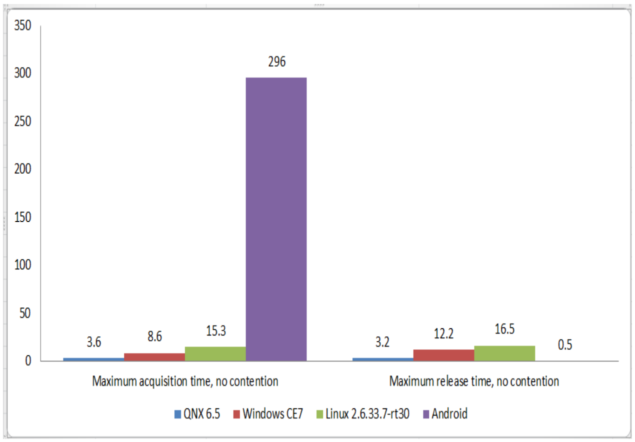


Figure 7b: Mutex <u>maximum</u> acquire-release time: no-contention case

Average times are not significant for this test because we are using a 13 MHz timer, which means that the measurement resolution is 0.077 µs, which is greater than the differences between OSs. The maximum value will depend largely on the clock tick duration.

4 Detailed Comparison

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 four tested OSs.

Clock interrupt duration	Average	Maximum
QNX Neutrino 6.5	2 µs	6.5 µs
Windows Embedded Compact 7	8.5 µs	12 µs
Linux RT 2.6.33.7	14 µs	25 µs
Android Linux 3.0.4	30 µs	300 µs

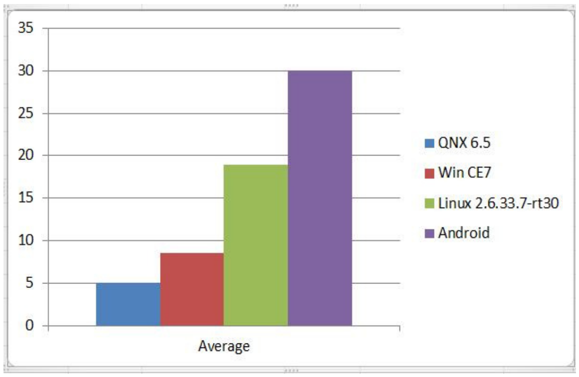


Figure 8a: ***Average*** clock interrupt duration

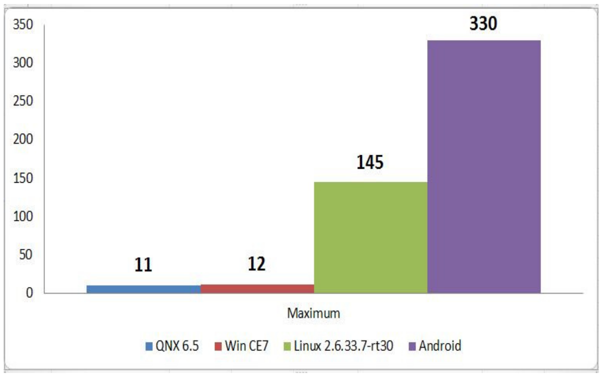


Figure 8b: ***Maximum*** clock interrupt duration

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 RTOSs (QNX Neutrino and Windows CE) perform far better than the Linux variants.

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 becomes active?**
- **If a thread is created with a higher priority than the creating thread, does this new thread become activate immediately?**

QNX Neutrino 6.5.0	Windows Embedded Compact 7	Linux 2.6.33.7-rt30	Android Linux 3.0.4
Successfully passed this test	Successfully passed this test	Successfully passed this test	Successfully passed this test

Table 6: Results for the thread creation test

QNX Neutrino and Windows CE7 passed this test successfully without any problems.

However, in both Linux variants 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 changing priorities at run-time is equivalent to dynamically creating and deleting threads, something that should not be done in a real-time system.

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 Neutrino 6.5.0	Windows Embedded Compact 7	Linux 2.6.33.7-rt30	Android Linux 3.0.4
Successfully passed this test	Passed ⚠	Passed ⚠	Passed ⚠

Table 7: Results of the round robin test

⚠ Note that:

- For the **Linux** and **Android** schedulers, the initial time slice of a created thread is 10 times greater than other slices (1second instead of the default 100milliseconds (ms)).
- Similar behavior was observed for **Windows Embedded Compact 7**: the initial time slice of a created thread is 10 times greater than other slices (100 milliseconds (ms) instead of 10 milliseconds where 1ms = 1tick).

Since dynamic thread creation and the use of different threads with the same priority is a poor practice in real-time systems, we assumed that real-time projects would avoid these practices. Given this assumption, we discounted these issues when we determined the final scores of the OSs we evaluated.

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		Windows CE7		Linux RT		Android	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Thread switch latency, 2 threads	0.6	6.3	2.7	15.3	3	21.4	7.9	317
Thread switch latency, 10 threads	0.6	3.9	3.1	15.1	3.3	28.6	8.4	321
Thread switch latency, 128 threads	1.2	8.2	5.5	17.7	5.3	29.1	13.2	363
Thread switch latency, 1000 threads	1.9	21.5	6.9	21	6.6	31.3	14.3	62.8

Table 8: Thread switch latency in µs between x threads

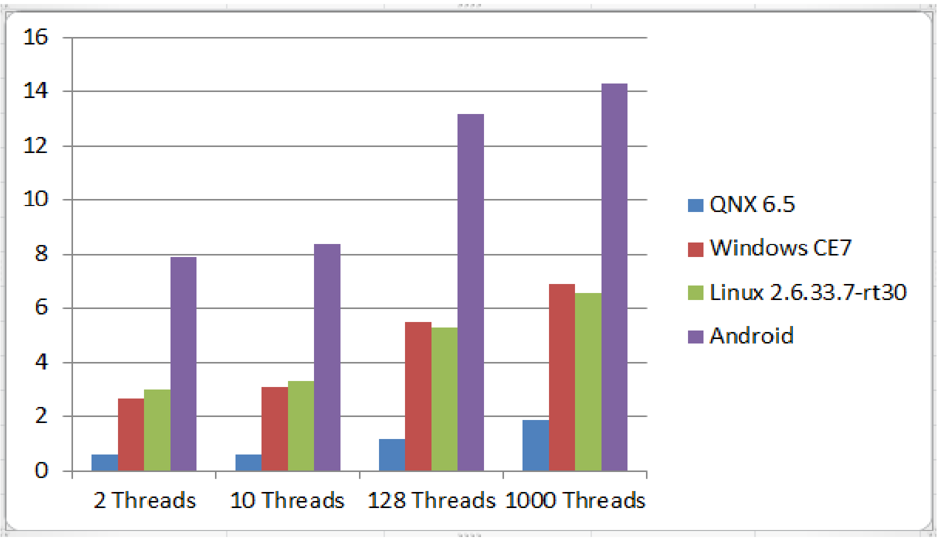


Figure 9a: <u>Average</u> switch latency between x threads, in µs

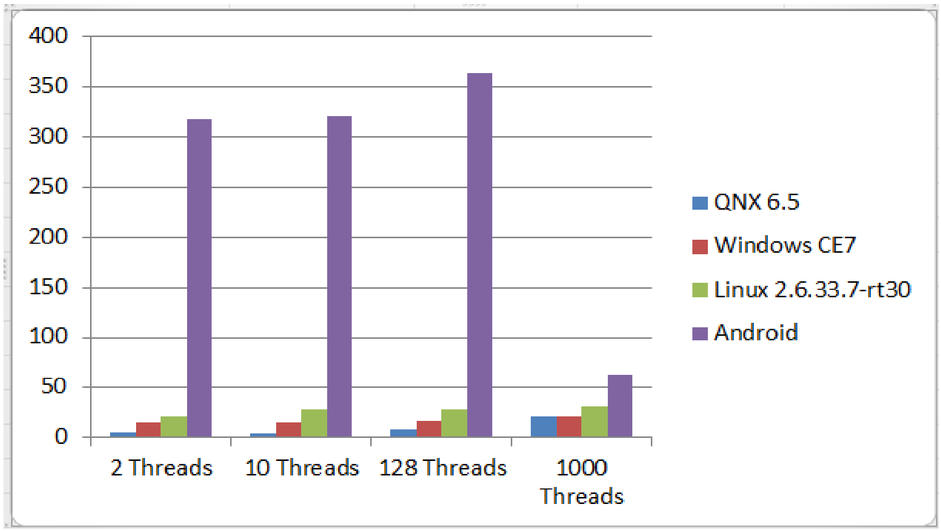


Figure 9b: <u>Maximum</u> switch latency between x threads, in µs

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. Interestingly, in the test with 1000 threads, the switch latency for the Android Linux OS does not follow the pattern of the other tests with the OS. We believe that this anomoly is due to our not catching the long clock tick during this particular test, but that a longer test would catch it and the anomoly would be corrected.

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, and therefore blocks.**

In the scenarios where the thread actually runs (2, 3), the creation time is the duration elapsed between the system call creating the thread and the moment the created thread is activated. For the “never run” scenario, the creation time is the duration of the system call.

Test		QNX		Windows CE7		Linux RT		Android	
		Avg	Max	Avg	Max	Avg	Max	Avg	Max
Never run	Thread creation	41.8	53.3	102	173	37.7	91.2	7.9	317
	Thread deletion	34.4	117	100	139	103.7	187.7	7.8	51.1
Run and terminate	Thread creation	41.7	55.9	111	197	79.3	146.6	183.4	532
	Thread deletion	3.1	23	3.3	14.1	3.3	33.2	6.8	594
Run and block	Thread creation	41.5	56.3	110	189	79.4	171.3	N.A	N.A
	Thread deletion	35.4	116.5	101	133	83.1	118.1	N.A	N.A

Table 9: Thread creation and deletion in µs

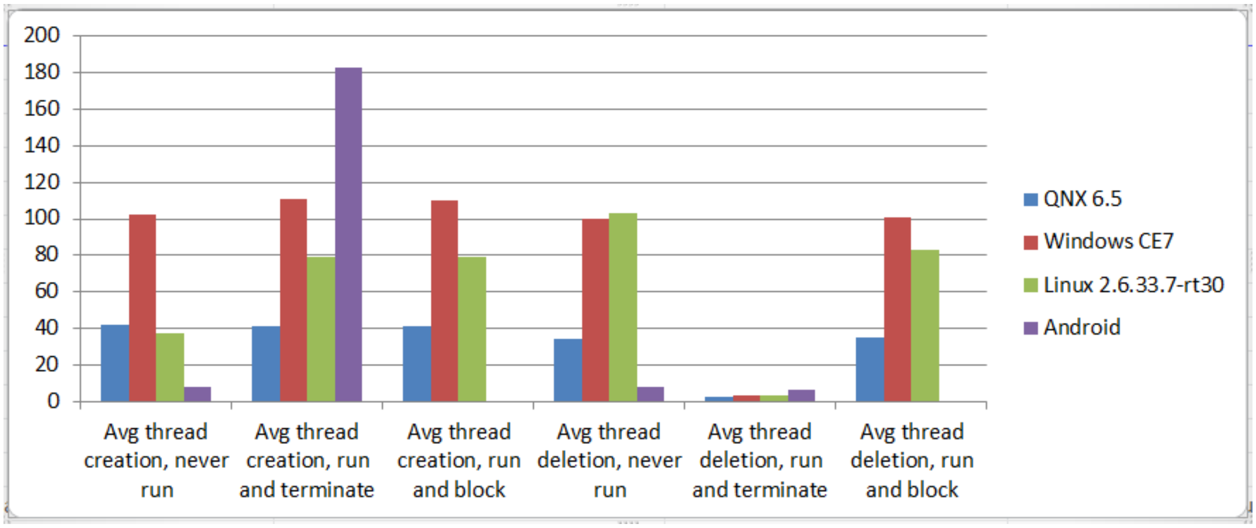


Figure 10a: Average thread creation and deletion times (µs) in different scenarios

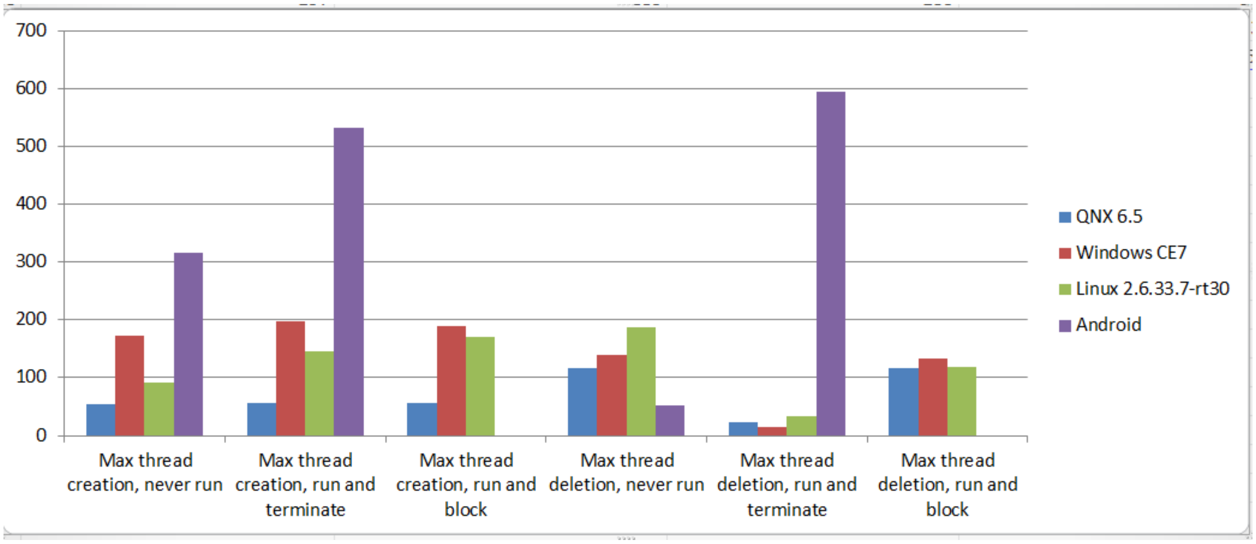


Figure 10b: Maximum thread creation and deletion times (µs) in different scenarios

Android Linux, which uses its own Bionic C library, behaves here very differently than the OSs with the glibc libraries. With Android Linux, thread creation and thread deletion is deferred and left to a management system using a manager thread that runs at a priority different than that of the thread to be deleted. Due to this behaviour, we could not run our scenario 3 test.

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 Neutrino 6.5.0	Windows Embedded Compact 7	Linux 2.6.33.7-rt30	Android Linux 3.0.4
The semaphore behaves correctly as a protection mechanism	The semaphore behaves correctly as a protection mechanism	The semaphore behaves correctly as a protection mechanism	The semaphore behaves correctly as a protection mechanism

Table 10: Semaphore behaviour results

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 Neutrino 6.5.0	Windows Embedded Compact 7	Linux 2.6.33.7-rt30	Android Linux 3.0.4
Successfully passed this test	Successfully passed this test	Successfully passed this test	Failed to pass this test

Table 11: semaphore release mechanism testing results

Android Linux did not pass this test because the Bionic C libraries use an un-prioritized FIFO on a semaphore release. Hence, the first thread blocked by the semaphore is the first thread released, no matter what its priority compared with other threads blocked on the semaphore. Although this behavior works as a protection mechanism, it is inappropriate for any real-time system.

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		Windows CE7		Linux RT		Android	
		Avg	Max	Avg	Max	Avg	Max	Avg	Max
Semaphore is used	Creation time	1.5	17.5	2.5	18.5	<0.1	17.5	<0.1	19.9
	Deletion time	1.5	13.5	2.6	14.2	<0.1	14.9	<0.1	0.7
Semaphore is never used	Creation time	1.5	26.3	2.4	13.4	<0.1	14.8	<0.1	42.3
	Deletion time	1.6	14.2	2.6	19.4	<0.1	14.7	<0.1	22.3

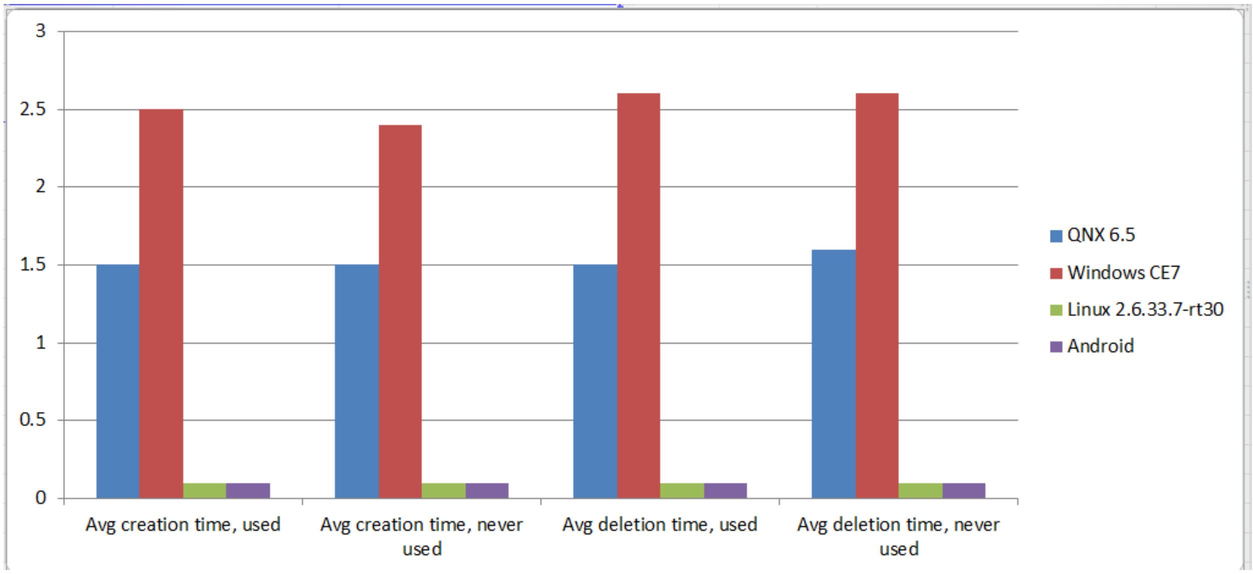


Figure 11a: Average creating and deleting times (µs) of a Semaphore in different cases

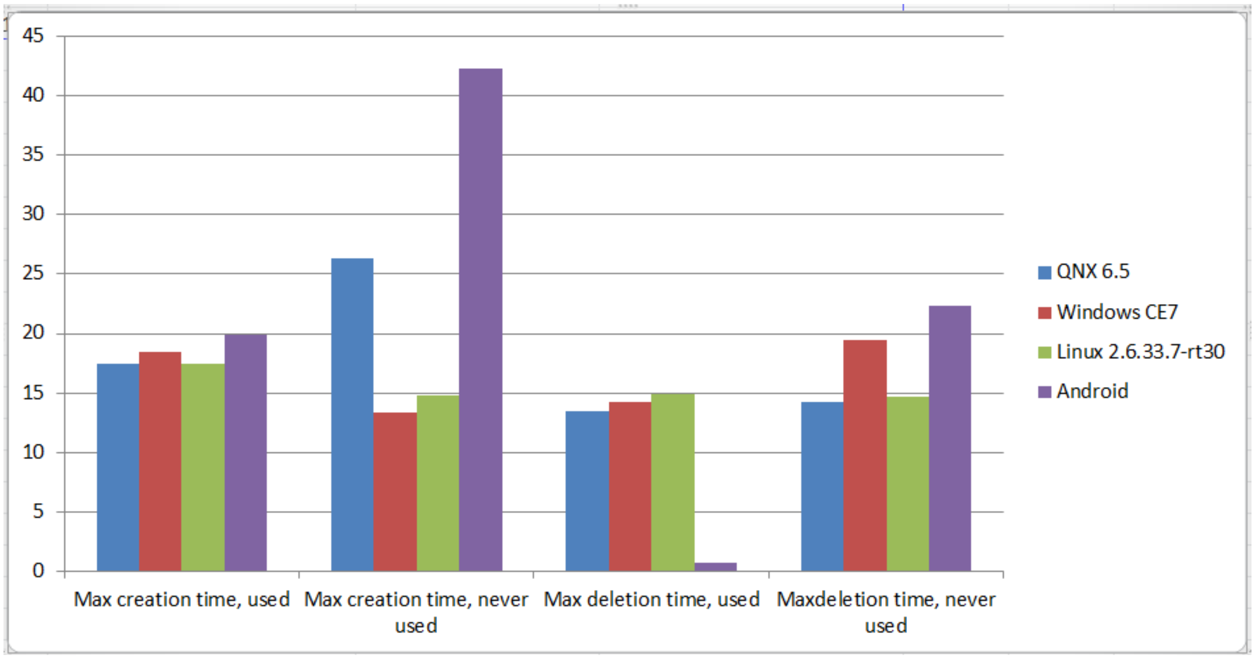


Figure 11b: Maximum creating and deleting times (µs)of a Semaphore in different cases

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		Windows CE7		Linux RT		Android	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Semaphore acquisition time, no contention	1.2	10.1	2.2	17.9	0.2	25	0.1	29.6
Semaphore release time, no contention	1.2	10.2	2	11.1	0.2	17.5	0.1	14.7

Table 12: Acquire release timings in the non-contention case

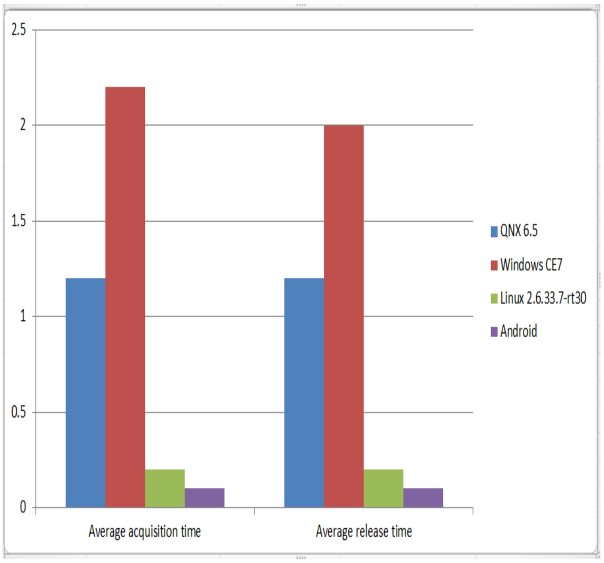


Figure 12a: Semaphore <u>average</u> acquire-release time: <u>no contention</u>

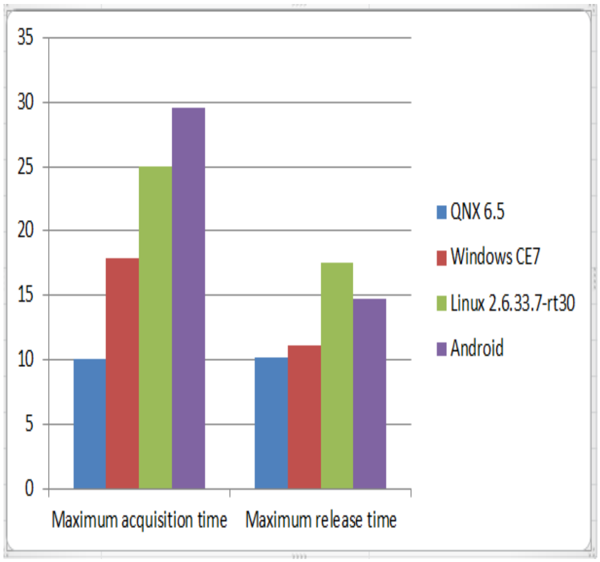


Figure 12b: Semaphore <u>maximum</u> acquire-release time: <u>no contention</u>

Both QNX and CE7 perform a round trip to the kernel in these cases (like that required for named semaphores) while the Linux variants use atomic instructions and do not need a round-trip.

Note, however, that semaphores are much more appropriate way for signaling threads than for use as a protection mechanism. Indeed, semaphores do not have the concept of ownership and thus cannot be used to prevent priority inversions. If protection is required, mutexes should be used instead.

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 5] for QNX Neutrino, [Doc 7] for Linux RT, [Doc 9] for Windows Embedded Compact 7 and [Doc 10] for Android Linux.

The test is conducted as follows:

- We create a semaphore with count zero: so it will block on acquire.
- We create 128 threads with different priorities. The creating thread has a lower priority than the threads being created.
- When a thread starts execution, it tries to acquire the semaphore; but as the semaphore is taken, the thread stops and the kernel switches 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". This time includes the thread switch time.
- Thread creation takes some time; so the time between each measurement point is large compared with most other tests.
- After the last thread is created and is blocked on the semaphore, the creating thread starts to release the semaphore, repeating this action the same number of times as the number of blocked threads on the semaphore.
- We start timing from the moment the semaphore is released, which in turn activates the pending thread with the highest priority, which stops the timing. Again, the thread switch time is included in the measurement.

Test	QNX		Windows CE7		Linux RT		Android	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Semaphore **acquisition** time, contention	3.8	62.3	7.5	110	7.6	260	11.5	68.9
Semaphore **release** time, contention	3.7	28.5	9.6	34.8	13.5	303	211	727

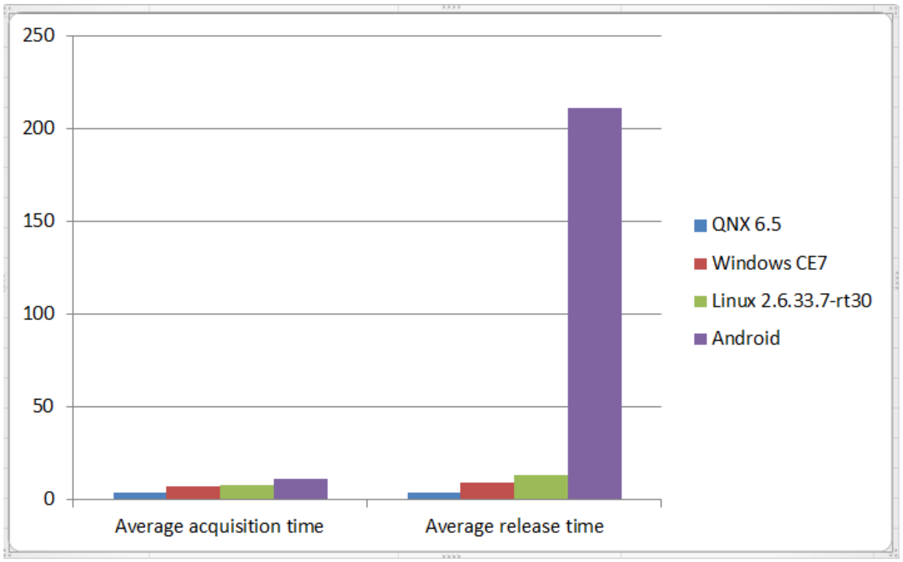


Figure 13a: Semaphore average acquire-release time: Contention

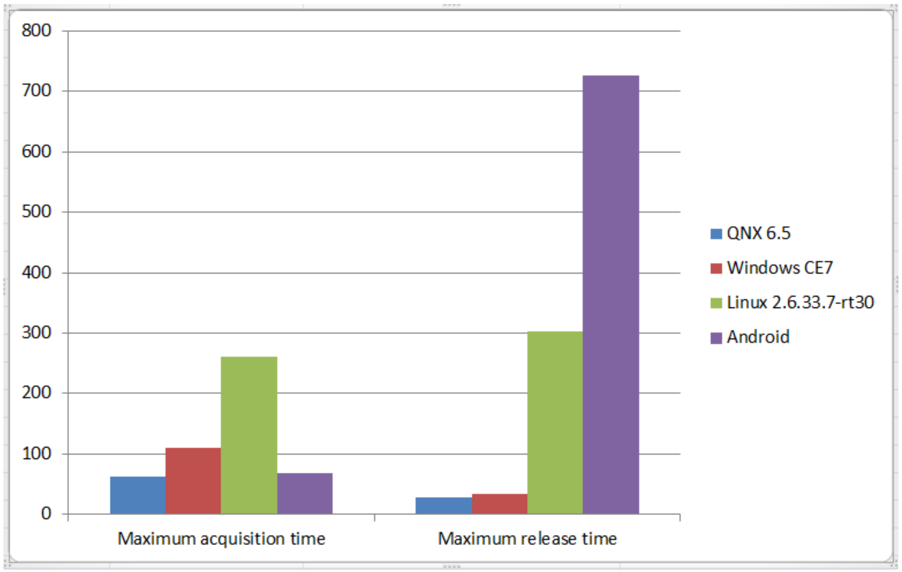


Figure 13b: Semaphore maximum acquire-release time: Contention

Both Linux RT and Android exhibited some strange behaviour: on Linux RT, we noticed that the clock tick interrupt duration time increases on release, influencing the worst case.
For Android, the semaphore release time increases dramatically when multiple threads are blocked on it. This increase explains why the average release time is so high, if we remember that we ran our tests with up to 128 threads being blocked by the semaphore).

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 Neutrino 6.5.0	Windows Embedded Compact 7	Linux 2.6.33.7-rt30	Android Linux 3.0.4
Priority inversion is prevented as expected	Priority inversion is prevented as expected	Priority inversion is prevented as expected	Failed to pass this test

Table 13: results of priority inversion avoidance mechanism

4.4.2 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		Windows CE7		Linux RT		Android	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Mutex **acquisition** time, no-contention	0.2	3.6	<0.1	8.6	0.2	15.3	<0.1	296
Mutex **release** time, no-contention	0.2	3.2	<0.1	12.2	0.2	16.5	<0.1	0.5

Table 14: Results of the mutex acquire-release timing in no-contention case, in µs

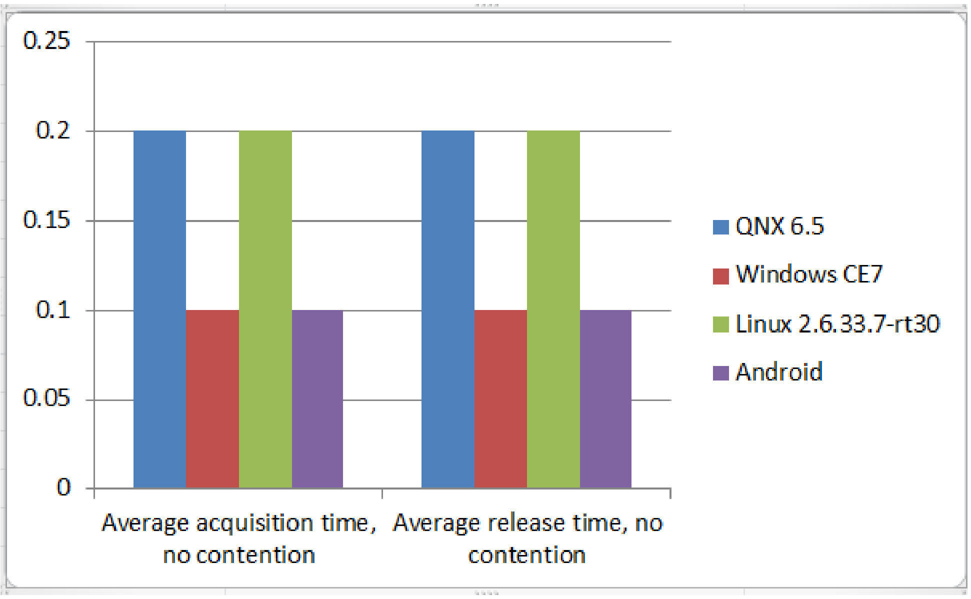


Figure 14a: Mutex average acquire-release time: no-contention

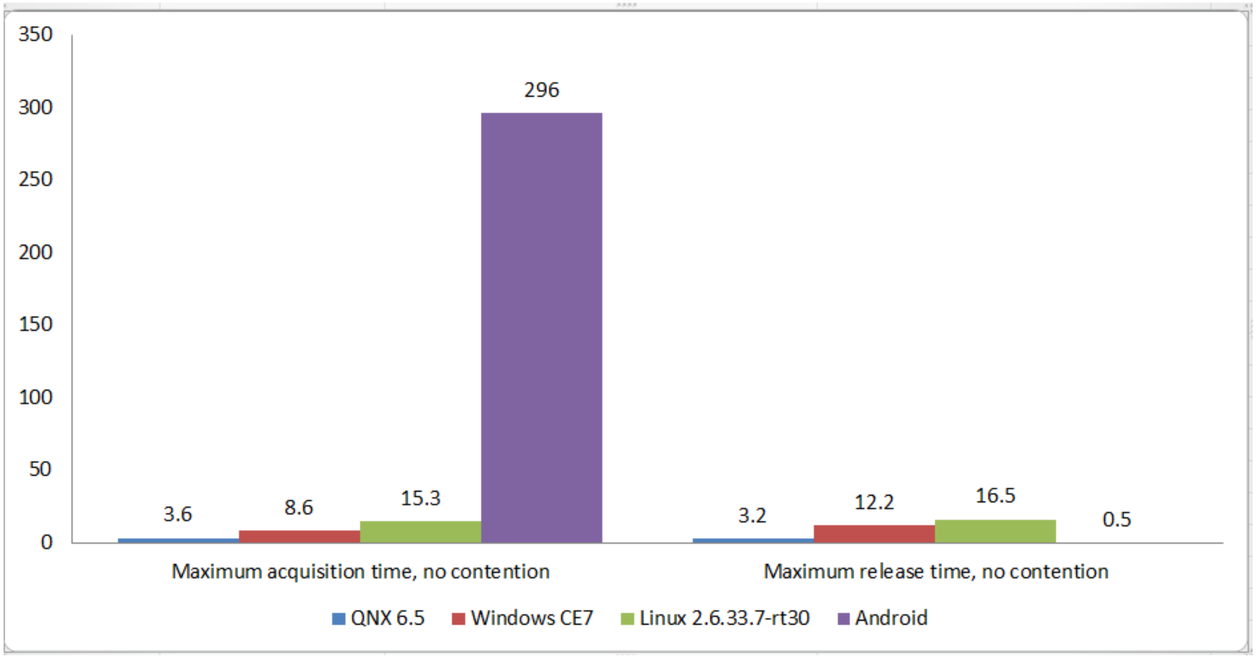


Figure 14b: Mutex maximum acquire-release time: no-contention

The average acquire-release time differences are too small to be measured with the 13MHz timer we used. However, the maximum values were measureable: Figure 14b presents the clock tick durations, in microseconds (µs).

Note that for the Android Linux OS release time, no clock tick was caught. We believe that if a clock tick had been caught, the value for the release time would have been the same as the maximum acquisition time.

4.4.3 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		Windows CE7		Linux RT		Android	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Mutex **acquisition** time, contention	2.2	7.2	5.3	16.6	10.3	32.4	11	72.6
Mutex **release** time, contention	3.3	13.5	5.8	21.9	25.9	55.1	12.8	320

Table 15: Results of the mutex acquire-release timing in contention case, in µs

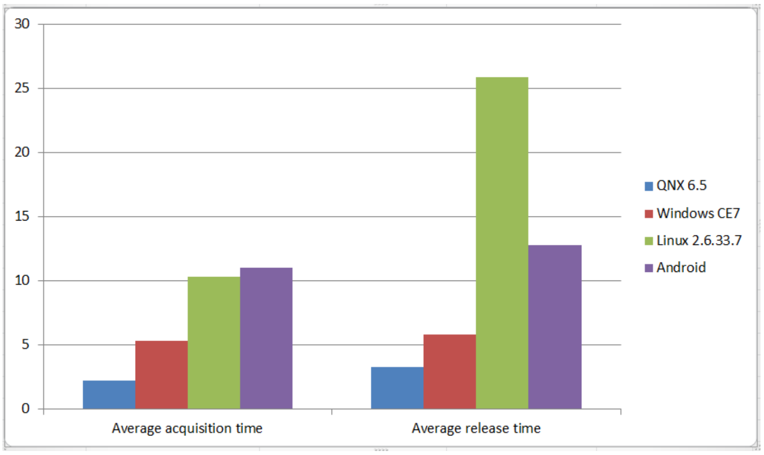


Figure 15a: Mutex average acquire-release time: contention

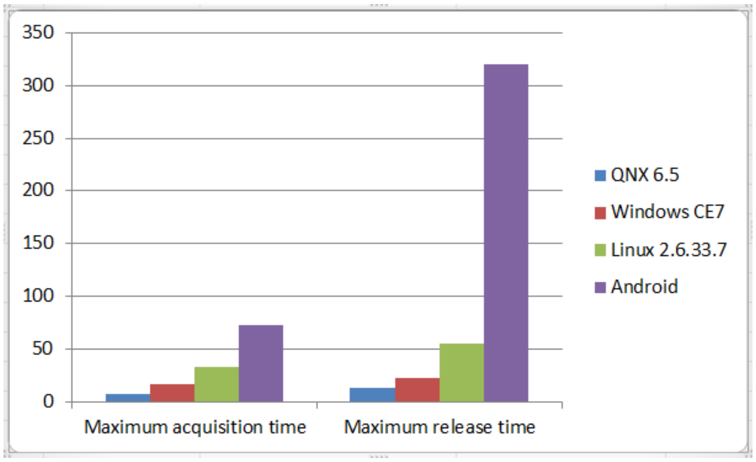


Figure 15b: Mutex maximum acquire-release time: contention

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, 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.

4.5.1 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		Windows CE7		Linux RT		Android	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Interrupt dispatch latency	0.5	2.6	2.5	14.4	0.8	14.9	1.9	30.9

Table 16: interrupt latency results in µs

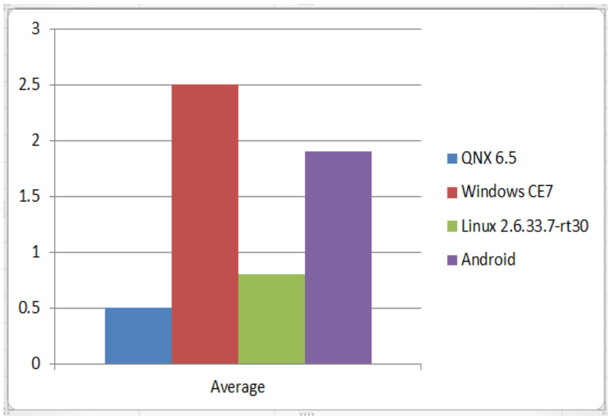


Figure 16a: Interrupt average dispatch latency

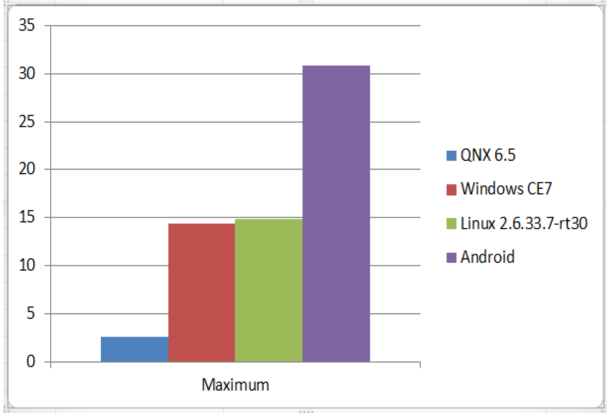


Figure 16b: Interrupt maximum dispatch latency

For average dispatch latency, QNX Neutrino showed very good results, as did Linux RT. Windows CE7 and Android Linux did less well.
However, for real-time systems, average dispatch latency is less important than maximum dispatch latency, or worst case. Of course there is no easy way to set an upper limit of the worst case. Using a statistical approach, you just need more samples to have a more accurate view. That's the reason why we run a long duration interrupt test. Hence, the maximum sustained interrupt rate is the most important test (section 4.5.4) for the real-time behavior.

A note of caution about interpreting results for these tests:

- In Windows CE, the interrupt handler will immediately launch an interrupt thread and mask the interrupt till the thread has finished its work. As a result, the interrupt is handled already in thread context. This increases a bit the latency, but improves the way you can prioritize things.
- This is also the case in Linux RT when using the `hard irq` option. So for this test, we used a non-delayed interrupt in Linux RT (thus, we avoided using the interrupt thread).

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		Windows CE7		Linux RT		Android	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Dispatch latency from interrupt handler	0.6	2.6	2.6	11.5	0.9	1.2	1.9	14.7

Table 17: interrupt dispatch latency in µs

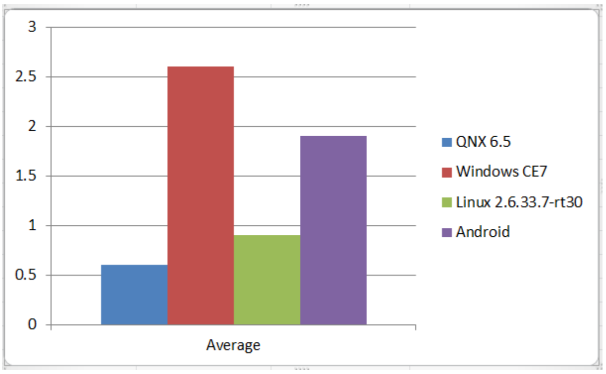


Figure 17a: Average dispatch latency from interrupt handler

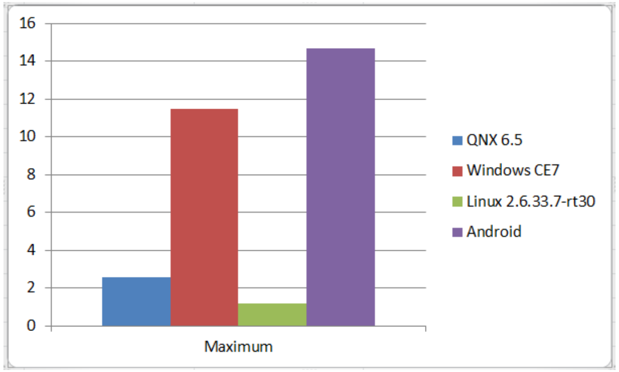


Figure 17b18: Maximum dispatch latency from interrupt handler

As discussed before, Windows CE uses an interrupt thread. So handling an interrupt involves a thread switch. At first glance, the Android latency doesn’t look so bad. However this is only a small test run. When running an endurance test, this quickly changes.

Note that the results for the maximum dispatch latency depend on whether or not the test catches timer interrupts. Note also that Windows CE7 uses an interrupt thread, as discussed above.

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 and Android Linux, a thread is blocked by using an `ioctl` call, and is released in the kernel module upon the interrupt.
- For Windows CE7, the interrupt handler is already running in a thread. We did not perform the “interrupt to thread latency test” on this OS as it involves a simple thread switch and is comparable with the semaphore release time.

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		Windows CE7		Linux RT		Android	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Latency from ISR to waken-up thread	1.2	6	N.A	N.A	8.1	13	29.5	300

Table 18: interrupt to thread latency, in µs

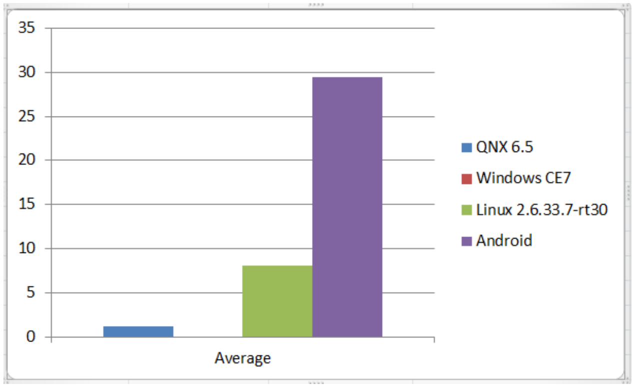


Figure 18a: Average and Maximum latency from ISR to waken-up thread, in µs

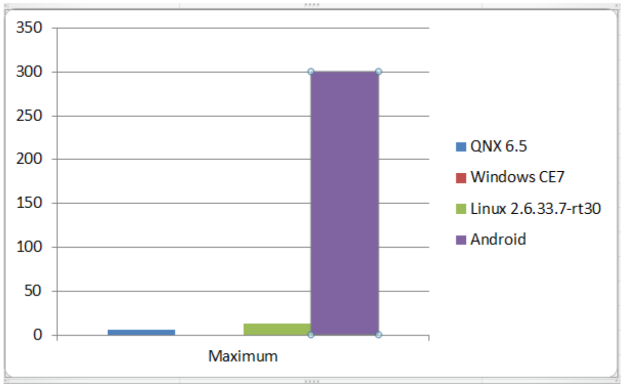


Figure 18b: Average and Maximum latency from ISR to waken-up thread, in µs

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. Nevertheless, we observed clear differences in the performance of the OSs we tested. In order to not miss any interrupts, Linux (with the RT patch) requires six (6) times more time between interrupts than does QNX Neutrino. Android Linux needs almost 18 times more time. Windows CE7 requires 26 µs which is not a far value from QNX Neutrino who had the best time in handling interrupts.

Test	QNX	Windows CE7	Linux RT	Android
Minimal interrupt period required not to lose any of the 1 billion generated interrupts.	**19***	**26**	**43**	**410***

* This value is not mentioned in the QNX technical report [Doc 5]. We just have there the time for 1 billion interrupts. As we are comparing the interrupt period of 100 million interrupts, we retrieved this value (19) from our tests database.

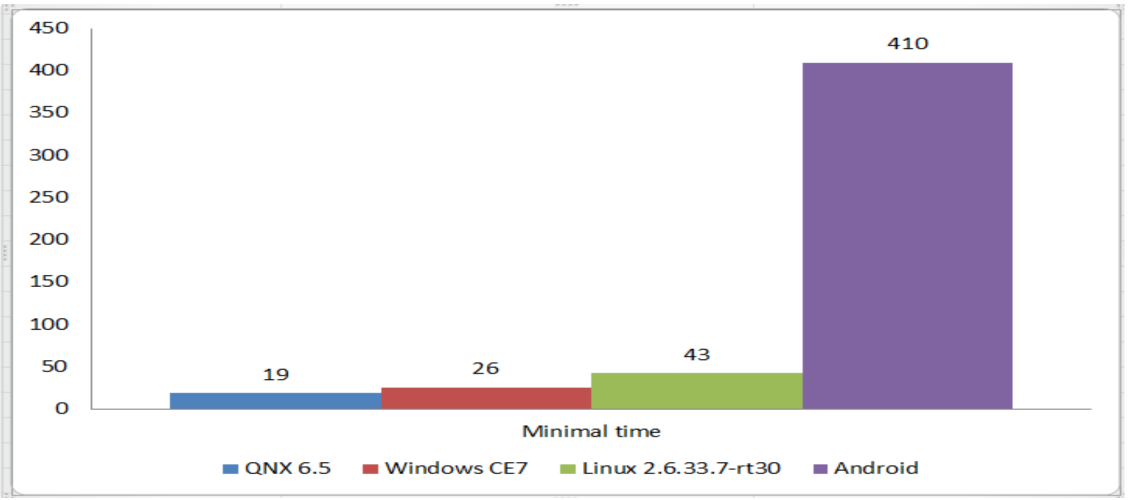


Figure 19: Minimal interrupt period required not to lose any of the 1 billion generated interrupts.

5 Conclusion

Our first conclusion is that both traditional RTOSs: QNX Neutrino and Windows CE7 can be qualified as true real time operating systems, out-of-the-box.

Linux with the RT_PREEMPT patch can also be qualified as RTOS, though for this OS the user must take care to use a correct kernel configuration, both at build time and at run time.
However, the behavior and performance gap that separated Linux RT from both QNX Neutrino and Windows CE7 do not make Linux RT a serious contender as a real-time OS.

Finally, our observations of the Android Linux, which diverges from the “vanilla” Linux kernel and uses other C libraries, such as the Bionic library, used in Google Android, are that this OS lacks the features and performance required to guarantee that it meets real-time deadlines. Indeed, Android Linux was developed with other requirements, makes no claims to keeping real-time deadlines, and should be used in the type of environments for which it was intended: smartphones and tablets.

6 Related documents

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

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 report
This document presents the qualitative discussion of the QNX OS
EVA-2.9-OS-QNX-65 Issue: 4.1 Sept 8, 2011

Doc.5 QNX technical evaluation report
This document presents the results of evaluating QNX on ARM platform
EVA-2.9-TST-QNX-65-ARM-01 Issue: 3.2 Sept 7, 2011

Doc. 6 Linux theoretical evaluation report
This document presents the qualitative discussion of the Linux OS
EVA-2.9-OS-LINUXRT_2.6.33.7.2-rt30 Issue: 1.07 May 30, 2011

Doc.7 Linux technical evaluation report
This document presents the results of evaluating Linux on ARM platform
EVA-2_9-TST-LINUXRT_2_6_33_7_2-rt30-ARM Issue: 3 Feb 17, 2012

Doc. 8 Windows Embedded Compact 7 theoretical evaluation report
This document presents the qualitative discussion of the Windows Embedded Compact 7 OS
EVA-2.9-OS-CE-7-A03 Issue: 2.1 Sept 19, 2011

Doc.9 Windows Embedded Compact 7 technical evaluation report
This document presents the results of evaluating Windows Embedded Compact 7 on ARM platform
EVA-2.9-TST-CE7-ARM Issue: 5 Feb 18, 2012

Doc.10 Android technical evaluation report
This document presents the results of evaluating Android on ARM platform
EVA-2_9-TST-ANDROID-3_0_4-ARM Issue: 2 Feb 17, 2012

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	Jan 17, 2012	ALL	Initial report
2.2	Feb 12,2012	ALL	Insert feedback
2.3	Feb 13, 2012	Some	Change tables’ headers
3	March 2,2012	All	Final version