



Q•Kernel

Thread-Metric RTOS Test Suite

Version 4.0-1775

Q•Kernel is a product of Quasarsoft Ltd.

Disclaimer

The information in this document is subject to change without notice. While the information herein is assumed to be accurate, Quasarsoft Ltd. (the manufacturer) assumes no responsibility for any errors or omissions.

The author makes and you receive no warranties or conditions, express, implied, statutory or in any communications with you. The manufacturer specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

In no event shall the manufacturer, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

Copyright notice

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the manufacturer. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license. If you have received this product under a Demo license for evaluation, you are entitled to evaluate it, but you may under no circumstances use it in a product. If you want to do so, you must obtain a fully licensed version from the manufacturer.

© 2007-2010 Quasarsoft Ltd

Trademarks

Names mentioned in this manual may be trademarks of their respective Companies. Brand and product names are trademarks or registered trademarks of their respective holders.



Quasarsoft Ltd
312-5th Avenue Suite No. 354
Cochrane Alberta T4C 2E3
Canada
Tel. +1 (403) 450 3482
www.quasarsoft.com

About this document

This document describes the Thread-Metric benchmark suite for **Q-Kernel** version V3.5-1523.

The Thread-Metric benchmark suite is a freely-available set of benchmarks that measures many aspects of RTOS performance, helping developers identify the bottlenecks in the real-time performance of their applications. Criteria such as interrupt response, context-switching, message passing, thread scheduling, memory allocation, and synchronization are particularly important for microcontroller-based designs where efficiency and a small, fast RTOS makes a significant difference. The Thread-Metric benchmark suite source code is available for free download from <http://www.embedded.com/code/2007code.htm>

The Thread-Metric benchmark suite is written by Express Logic, Inc, 11423 West Bernardo Court, San Diego, CA USA



1.	Description of the test implementation	5
1.1.	Hardware.....	5
1.2.	Software.....	5
2.	Test results and implementation	6
2.1.	Cooperative Scheduling	7
2.2.	Pre-emptive Scheduling	8
2.3.	Interrupt Processing.....	8
2.4.	Interrupt Pre-emptive processing.....	8
2.5.	Message Processing.....	8
2.6.	Synchronization processing	9
2.7.	Memory allocation.....	9

1. Description of the test implementation

The Vendor (in this case Quasarsoft Ltd.) needs to implement a porting layer to make the benchmark working. Several aspects of the implementation are describe in the next chapters.

1.1. Hardware

The tests are executed on the Explorer-16 board manufactured by Microchip. A PIC 24HJ256GP610 is placed on the board for testing. The board is connected to a Real-ICE that loads the program on the chip. The microprocessor run at 80 MHz producing 40MIPS.

1.2. Software

Quasarsoft has implemented the porting layer with version 4.0-1770 of **Q-Kernel**. The porting layer (tm_proting_layer.c) contains all code with the exception of raising the interrupts. Raising the interrupt (_INT0IF=1) is implemented in the files tm_interrupt_processing_text.c and tm_interrupt_preemption_processing_text.c.

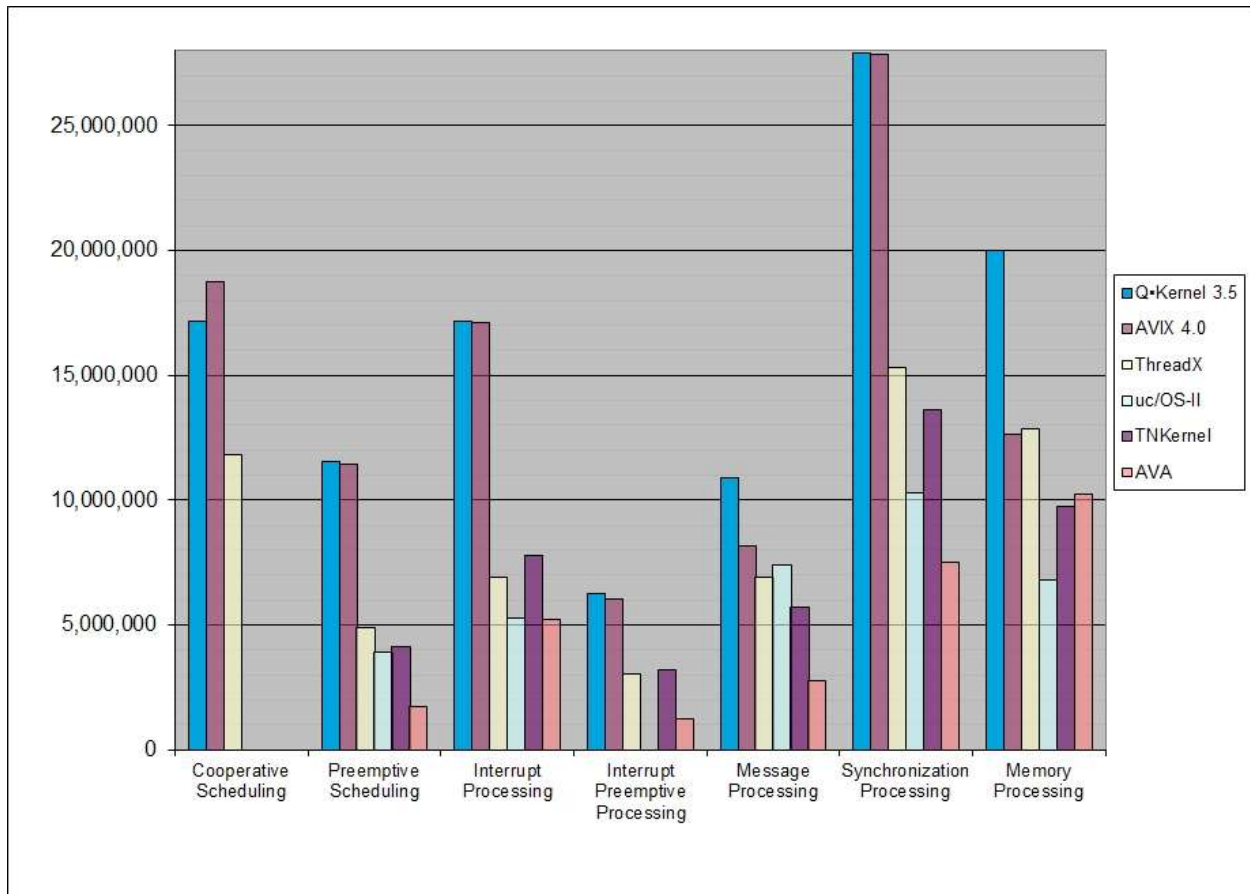
The porting layer can execute 7 different tests by changing the test number in the file and compile and run the test.

The project uses the QKernel16Fst-elf.a library that contains the objects that are compiled with version V3.23 of the C30 compiler with only one compiler switch -O3. This switch compiles the code to achieve the best performance. It uses in-line functions and other algorithms that create fast code but also results in the largest use of flash.

2. Test results and implementation

Q-Kernel is one of the best performing RTOS's, because it is tick-less and it uses the segmented interrupt architecture. The results are in the following table. It should be clear that **Q-Kernel** and AVIX are outperforming all other products. Both implement the segmented interrupt architecture. FreeRTOS is not in the list because the license agreement prohibits us from publishing benchmarks. Please check it yourself and it will be clear why they don't allow us to publish their performance numbers.

	Cooperative Scheduling	Preemptive Scheduling	Interrupt Processing	Interrupt Preemptive Processing	Message Processing	Synchronization Processing	Memory Processing
Q-Kernel 3.5	17,141,251	11,559,613	17,141,260	6,282,177	10,908,076	27,904,359	19,998,144
AVIX 4.0	18,730,514	11,460,380	17,125,013	6,023,870	8,151,857	27,878,435	12,618,419
ThreadX	11,847,800	4,870,885	6,918,050	3,052,151	6,928,383	15,337,354	12,863,624
uc/OS-II		3,909,085	5,259,998		7,387,612	10,293,318	6,814,817
TNKernel		4,138,692	7,784,052	3,180,224	5,722,266	13,623,702	9,745,907
AVA		1,724,948	5,207,762	1,260,190	2,761,154	7,514,799	10,235,182



While both AVIX and **Q-Kernel** are based on the segmented interrupt architecture, differences between the two products are significant. The scheduling engine of **Q-Kernel** is more complex because it also has to schedule fibers. The scheduling engine has to check for fibers because it does not know that they are not used. AVIX does not support fibers so they don't have to be checked. **Q-Kernel** uses a number of linked lists for thread scheduling while AVIX uses a list for every priority. This improves the scheduling performance but requires more RAM and Flash. More variation in priorities will increase the memory footprint, therefore AVIX requires the developer to configure a maximum priority at the cost of the RAM foot-print. AVIX also uses more flash. The following list compares the size of the Thread-Metric suite between **Q-Kernel** and AVIX.

RTOS and library options	Flash size (words) TM program ¹
AVIX	8505
Q-Kernel with use of fast ² library	7418 (14% smaller than AVIX)
Q-Kernel with use of standard ³ library	6996 (18% smaller than AVIX)
Q-Kernel with use of small ⁴ library	6697 (21% smaller than AVIX)

While AVIX comes with only one library, **Q-Kernel** provides a standard library and two libraries optimized for speed and size.

Q-Kernel performs better even if **Q-Kernel** and AVIX require the same number of cycles to execute a function. The reason for the better performance is that **Q-Kernel** is not constantly interrupted by the RTOS Tick. **Q-Kernel** is tick-less and does not require cycles for the tick handling. AVIX uses a long tick-time (1000 μSeconds) for the test to minimize tick overhead. **Q-Kernel** is tick-less and has a theoretical fixed timing granularity of 1 cycle and a 1 μSeconds in practice. AVIX and other competitors allow shorter tick-times but will create more overhead. This overhead becomes more than 10% with a tick-time of 10 μSeconds.

2.1. Cooperative Scheduling

Q-Kernel can implement cooperative scheduling with threads, fibers or lightweight threads. Lightweight threads or fibers are a better solution for cooperative multi-threading and significant faster but Quasarsoft has implemented the test with threads to comply with the intentions of the Thread-Metric tests. This test is implemented with the **Q-Kernel** function qThrYield(). This function removes the current thread from the top of the ready list and moves the thread to the end of the list of threads with the same priority and switches the context.

Cooperative scheduling is included in the test suite but is hardly used in embedded applications. **Q-Kernel** is optimized for preemptive scheduling.

¹ This is the size in words and compiled with version 3.25 of the C30.

² The fast library provides the fastest code but also the largest flash footprint. This library is used in the test.

³ The standard library provides complete and constant error checking but creates a larger footprint.

⁴ The small library provides the smallest footprint and is just between fractional slower than the fast library.

2.2. Pre-emptive Scheduling

Pre-emptive scheduling is a type of scheduling where the thread is stopped at any possible time at any possible instruction and another thread is activated. In other words, a thread switch occurs.

Q-Kernel implements pre-emptive scheduling with the functions `qThrSuspend()` and `qThrResume()`. The function `qThrSuspend()` removes the thread from the ready list into a hibernate state and does a context switch. The function `qThrResume()` moves a thread out of the hibernate state into the ready list and switches the context if the top of the ready list contains a thread with a higher priority than the running thread.

2.3. Interrupt Processing

Interrupt processing is implemented with a C30 style interrupt handler and the functions `qSemAcquire()` and `qSemReleaseISR()`. One thread generates an interrupt. The interrupt handler calls the `tm_interrupt_handler()` and that function releases the semaphore. The thread that generated the interrupt acquires the semaphore.

The C30 style interrupt handler uses the shadow registers and does not save the PSVPAG because there is only one PSV window required. All **Q-Kernel** signaling function can be called from within an ISR so the release of the semaphore is done from within the ISR.

2.4. Interrupt Pre-emptive processing

In preemptive multi-threading, an interrupt can cause preemptive activity. In fact, an interrupt service routine preempts execution of code while it services the interrupt. However, while it simply returns back to the point of interruption, **Q-Kernel** could intercept and resume execution in another thread.

Interrupt pre-emptive processing is implemented with a C30 style interrupt handler and the functions `qThrSuspend()` and `qThrResume()`. One thread generates an interrupt. The interrupt handler calls the `tm_interrupt_preemption_handler()` and that function calls `qThrResume()`. This function move the thread from the waiting list into the ready list. **Q-Kernel** will become active at the end of the interrupt handler and will pre-empt the current thread and run the new thread.

The C30 style interrupt handler uses the shadow registers and does not save the PSVPAG because there is only one PSV window required. All **Q-Kernel** signaling function can be called from within an ISR so the resume of a waiting thread is done from within the ISR.

2.5. Message Processing

Q-Kernel has two implementations for sending and receiving messages. The most advanced method uses managed messages where **Q-Kernel** controls the life-time of a message and will do all memory management. This method allows variable sized messages and will move messages by reference and not by value. It is also possible to allocate and de-allocate message from interrupt handlers but the Thread-Metric test suite does not implement that.

The second implementation (pipes) send message by value and don't keep a use count. Because the second method complies more with the intentions of the

Thread-Metric tests we have used pipes. Our implementation of pipes is completely in assembler for the best performance.

While the Thread-Metric test suite calls the send and receive from only one thread the **Q-Kernel** functions implement the full spectrum of RTOS functionality like testing if a potential blocking function need to be called. The functions also contain critical section handling to synchronize thread access.

2.6. Synchronization processing

Q-Kernel has multiple mechanisms for synchronization. The mechanism that complies most with the intentions of the Thread-Metric tests are semaphores. Conceptually, a semaphore maintains a set of permits. Each qSemAcquire() blocks if necessary until a permit is available, and then takes it. Each qSemRelease() adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly.

While the Thread-Metric test suite calls the acquire and release functions from only one thread the **Q-Kernel** functions implement the full spectrum of RTOS functionality like blocking and critical section handling to synchronize thread access.

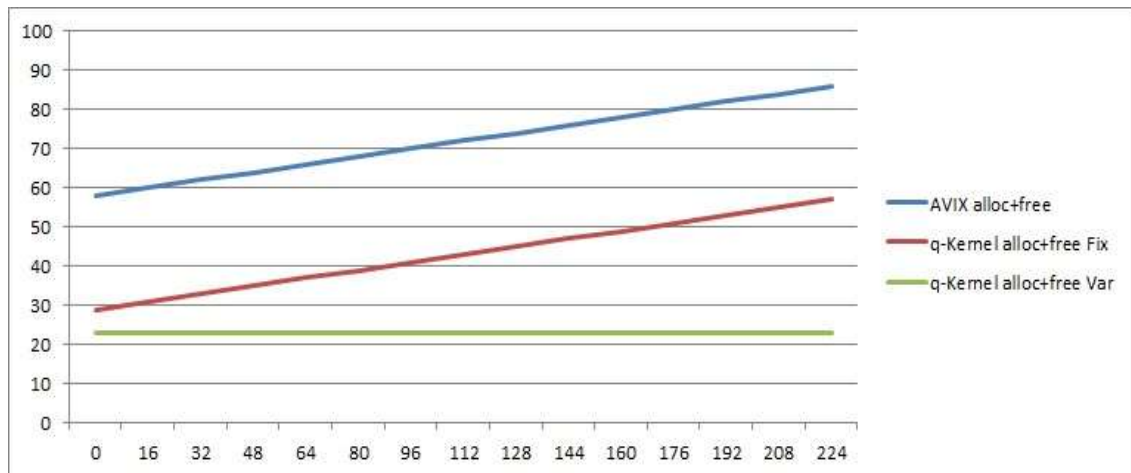
2.7. Memory allocation

Because **Q-Kernel** manages all its resources dynamically it requires a dynamic memory management system. While most competitors provide simple fixed size memory blocks **Q-Kernel** offers real dynamic memory allocation without external fragmentation called Variable Memory Blocks. This memory is organized in pools and can be accessed by size or by pool. **Accessing memory by pool is extremely fast and 100% deterministic.** **Q-Kernel** also provides two other memory allocation mechanisms, "Allocate Only Heap" and "Fixed Memory Blocks". Fixed Memory Blocks can be allocated and de-allocated from interrupt handlers.

The tests can be performed with fixed or variable memory blocks and the required code is included in the porting layer. The memory processing numbers are 19,998,144 for variable blocks and 18,180,129 for fixed memory blocks.

Some systems provide blocking functionality for memory allocation. The Thread-Metric test suite does not require this functionality and some vendors, including ThreadX, implement the test without blocking functionality by specifying a "NO_WAIT" parameter and will return an error if memory is not available. The **Q-Kernel** implementation will first try to allocate a block from the pool and if no memory block is available it will try to allocate a block from the "allocate only heap" and will extend the pool. The system will throw an error if everything fails, just like ThreadX. This behavior follows the more dynamic nature of the **Q-Kernel** memory management.

The following graph shows allocating and de-allocating of memory blocks from a 224 block pool. The X-Axis shows the number of memory blocks already in use and the y-axis show the number of cycles required for allocation and freeing a memory block. AVIX allocates and de-allocates the first block from the pool in 58 cycles. **Q-Kernel** allocates and de-allocates its first block from the fixed pool in 29 cycles while the variable pool only takes 23 cycles. The 180th block will take AVIX 80 cycles and **Q-Kernel** 23 cycles (variable pool) or 50 cycles (fixed pool).



Both products have the same degree of determinist-icy for memory blocks that can be allocated from interrupts, but **Q-Kernel** is much faster and also provides the developer with variable blocks that are even faster and 100% deterministic. Variable memory is the best option if interrupt handlers don't allocate memory. Variable memory blocks can be freed from interrupts by using the deferred function qMemFreeISR().

While the difference in raw performance is very large (23 cycles versus 58 cycles or about 2.5 times as fast) the TM suite only shows a difference of 1.6 times. This is because the test code itself adds 37 cycles. So the test takes 23+37=60 for **Q-Kernel** and 58+37=95 cycles for AVIX.