Q·Kernel

Thread-Metric RTOS Test Suite

Version 6.0-3343

Q·Kernel is a product of Quasarsoft Ltd.

License

Q-Kernel-Free is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License (version 3) as published by the Free Software Foundation **and modified by** the QuasarSoft Ltd. exception.

> The QuasarSoft Ltd. EXCEPTION
>
> You may not exercise any of the rights granted to You in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Program for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

Q-Kernel-Free is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the full license text at the following link <http://www.quasarsoft.com/license.html>

For the purpose of applying the license to this document, I consider "source code" to refer to this document source (.docx) and "object code" to refer to the generated file (.pdf).

Quasarsoft Ltd
312 5th Ave Bay 14
Suite 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 V6.0

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

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 6.0-3343 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_preemtion_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.

1.3. Results in spreadsheet

On our web-site you can find a spreadsheet with the calculation of all results and more information how the systems compare.

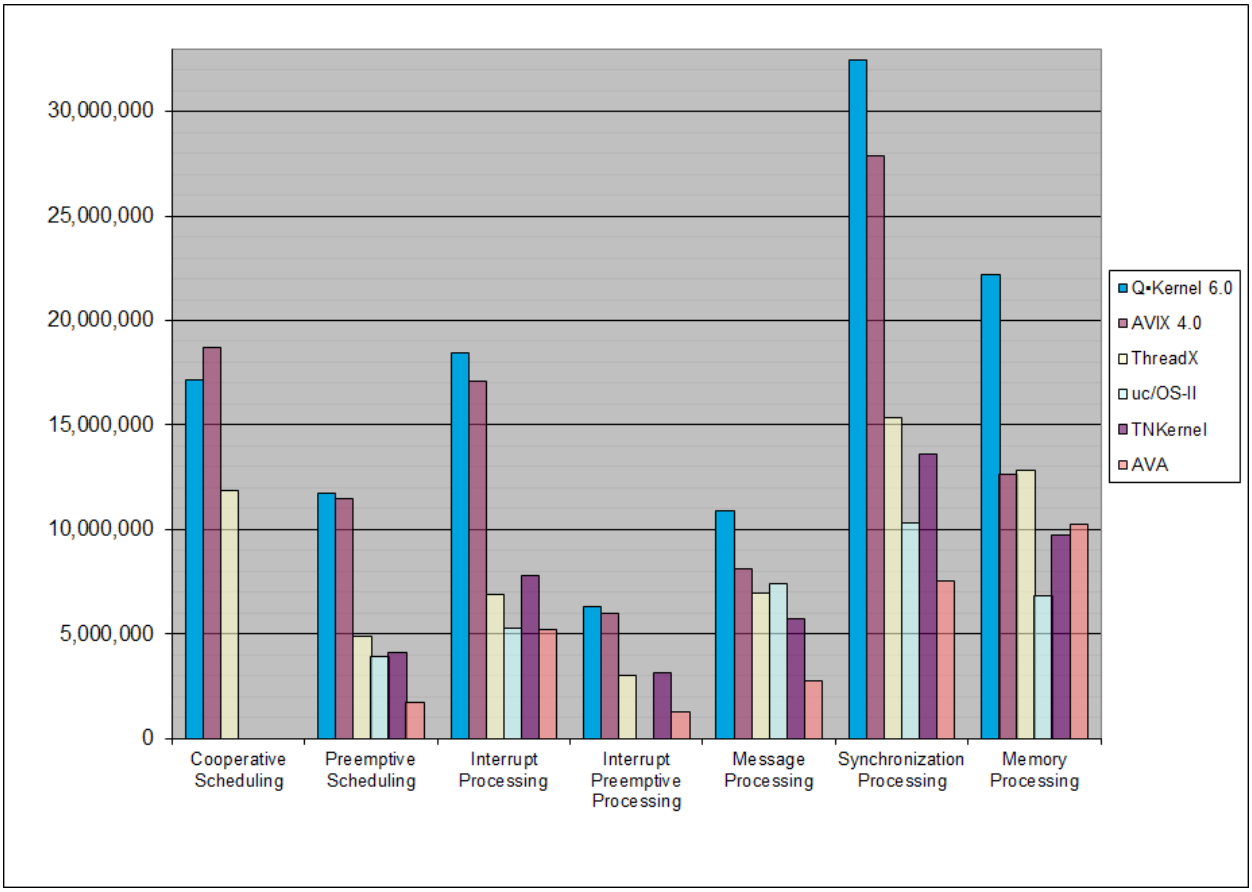
1.4. Participants

Q·Kernel is compared to AVIX, Thread-X, uc/OS-II, TNKernel and AVA. FreeRTOS™ is not in the list because the license agreement does not allow us to compare it. It is the only system that has this clause in their license agreement. We did the tests and it is pretty clear why that clause is there.

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* 6.0**	17,141,251	**11,741,503**	**18,461,234**	**6,315,823**	**10,908,076**	**32,431,922**	**22,221,851**
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	7418 (14% smaller than AVIX)

While AVIX can keep up with ***Q·Kernel*** (average 18% slower), the other systems are significant slower. The reason for this is that they are much older and use a different programming paradigm. Newer processors must keep their pipe-line full so calling is an expensive operations. ***Q·Kernel*** tries to execute as much as possible code inline. A good example is de-allocating memory. The function is only 8 cycles but calling and returning requires an additional 6 cycles. So the inline function requires 8 cycles compared to 14 cycles. That's not all the savings. If a functions is used the compiler cannot trust registers W0 to W7. The inline function only uses 2 register so the compiler has 6 more registers to optimize code around the de-allocation operation. This inline approach cost a bit more flash, 3 program words versus 8 program words. How is it possible that ***Q·Kernel*** is still smaller than its competitors? This is because ***Q·Kernel*** consists of about 200 small modules and only the modules that are used are linked into the end-results.

Some vendors have changed the code of the test software so it will fit their product better and produces better results. One example of this are code changes where the same functions is called from an ISR or a thread. This prevents testing the interrupt level and calling the right module. All ***Q·Kernel*** functions can be called from interrupt handlers or threads and no developer action is required.

Q·Kernel can do almost 800,000 context switches per second on a 40MIPS PIC24 which is 1.28 µSec per switch.

[1] This is the size in words and compiled with C30.

2.1. Cooperative Scheduling

Q·Kernel can implement cooperative scheduling with threads or fibers. 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.

Q·Kernel can do about 390,000 dual context switches per second on a 40MIPS PIC24 which is 1.28 µSec per switch.

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 qThrSuspendV4()[2] and qThrResumeV4(). The function qThrSuspendV4() removes the thread from the ready list into a hibernate state and does a context switch. The function qThrResumeV4() 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 procession is implemented with a C30/XC16 style interrupt handler and the functions qSemAcquireFast[3]() and qSemReleaseFast(). 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] We use Version4 functions because this functionality fits the test better. The thread resume and suspend functions of version 6 allows the developer to communicate between the two threads.

[3] We use “Fast” functions for optimized performance. These functions are functional the same as the “non-fast” functions but don’t do extensive parameter testing.

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 procession 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.

Q·Kernel can process more than 350,000 messages per second on a 40MIPS PIC24.

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 qSemAcquireFast() blocks if necessary until a permit is available, and then takes it. Each qSemReleaseFast() 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.

Q·Kernel can synchronize more than one million processing request per second on a 40MIPS PIC24.

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 22,221,851 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 and error if everything fails, just like ThreadX. This behavior follows the more dynamic nature of the ***Q·Kernel*** memory management.

Memory allocation and de-allocation is 100% deterministic if memory blocks are created ahead of time because this prevents dynamically extending the pool. This is how the test is executed.

Q·Kernel can allocate and free more than 740,000 memory blocks per second on a 40MIPS PIC24.

Memory allocation is one of the most used functions and Q·Kernel is more than 75% faster than the second best one.