Q▪Kernel
Feature Guide
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 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 features of ***Q·Kernel*** *(The new generation RTOS)* in more detail than the web-site. This guide is for all version of ***Q·Kernel.***

How to use this manual

The intention of this manual is to give you a good indication about the features of ***Q·Kernel*** and the differences between the MCU versions. For a more comprehensive description how to use ***Q·Kernel*** please read the ***Q·Kernel*** User guide.

1. Architecture 6
1.1. Segmented Interrupt Architecture 6
2. Architectural features 8
2.1. Dual-Mode RTOS 8
2.2. Low Power and Tick-less 9
2.3. Integrated Power Management 9
2.4. Never Disables Interrupts 9
2.5. Zero Interrupt Latency and No Interrupt Jitter 10
2.6. Hard Real Time 10
3. Feature and functionality 11
3.1. Threads, Fibers and Lightweight Threads 11
3.2. Memory management 11
3.3. Statistics and Thread & Fiber Tracking 12
3.4. Real-time Clock, Alarm and Timer functions 12
3.5. Centralized Error Handling 13
3.6. Scheduling 13
3.7. Minimize RAM usage 13
3.8. Advanced API 14
3.9. Threads 14
3.10. Fibers 16
3.11. Lightweight Treads 16
4. Managed Kernel Objects 17
4.1. Mutexes 17
4.2. Semaphores 18
4.3. EventSets 18
4.4. Pipes 19
4.5. Messages 19
4.6. Publish/subscribe functionality 20
5. Performance 22
5.1. Tick-less operation 23
5.2. Memory allocation 24
5.3. Context switching 25

© 2008-2012 Quasarsoft Ltd. qKernelFeatureGuide V4.0-1775

Introduction

Q·Kernel is a preemptive Real Time Operating System, or Kernel, specifically developed for a new generation of processors. ***Q·Kernel*** fully exploits the capabilities of those processors by implementing a unique segmented interrupt architecture, making it the fastest and most versatile RTOS. The architecture enables Dual-Mode capabilities not found in other Real Time Operating System.

What makes *Q·Kernel* Unique?

Q·Kernel has some unique features not found in any other Real-Time Operating System. The list of features is very complete, but a number of features really separate ***Q·Kernel*** from the competition.

- **Dual-Mode RTOS**

  The Dual-Mode features of ***Q·Kernel*** means that you can use the same RTOS for control applications and high dataflow and/or DSP applications.

- **Tick-less RTOS**

  Because ***Q·Kernel*** is tick-less the timing granularity is very fine, up to 1 µSecond and there is no tick to consume power.

- **Power management**

  Integrated power management helps the developer to design systems with low power consumption, without much effort. The power management combined with the tick-less feature produces an RTOS with the lowest power consumption in the market place.

- **Zero interrupt latency**

  Q·Kernel never disables interrupts, not for a single cycle. This allows any applications to use an RTOS because the interrupt latency is only determined by the underlying hardware.

- **Advanced interrupt stack**

  Q·Kernel uses an interrupt stack implementation that switches the stack before the user interrupt code is executed. This implementation is unique in the business and minimizes RAM usage significantly.

- **Advanced memory system**

  While most competitors provide the user with fixed memory blocks, ***Q·Kernel*** supplies the developer with an advanced memory systems that is deterministic but still flexible. The developer can allocate and de-allocate memory with different sizes without the risk of fragmentation.

- **Threads, Fibers and Lightweight threads**

  Most competing products provide threads and some provide lightweight treads. ***Q·Kernel*** is the only RTOS that provides the developer with threads, fibers and lightweight treads. This means that the developer can use the right tool for the right task and optimize speed and/or minimize RAM usage.

- **Best error handling in the business**

  Q·Kernel is the only RTOS that provides the developer with a centralized error management system and a ways to log errors in flash that always work.

1. Architecture

The architecture of ***Q·Kernel*** set it aside of most competitors. It uses the "Segmented Interrupt" architecture, is Dual-mode, tick-less, contains power management and never disables interrupts for real zero interrupt latency.

1.1. Segmented Interrupt Architecture

Most competitive systems are designed to disable interrupts during critical operations. This design is called the "Unified Interrupt Architecture". ***Q·Kernel*** is designed to never disable interrupts and this architecture is called the "Segmented Interrupt" architecture and makes losing interrupts a thing of the past. This unique architecture will never disable interrupts and the combination with fibers makes interrupt handling very fast. As a result ***Q·Kernel*** does not add a single cycle to the interrupt latency and facilitates seamless communication between Interrupt Service Routines, threads and fibers.

The interrupt processing prevents Interrupt jitter. Reducing the jitter is important for any RTOS, since they must maintain a guarantee that the execution of specific code will complete within an agreed amount of time. Only "Segmented Interrupt" architectures are able to guarantee jitter free operation.

In short the advantages are:

- Interrupt jitter is low or even zero for the highest interrupt priority.
- Interrupt latency the same as that of the underlying hardware.
- Very high interrupt rates. ***Q·Kernel*** running on a 40MHz PIC24H can handle more than 570,000 interrupts per second. The best performing "Unified Interrupt" architecture implementation (ThreadX) only handles 230,000 interrupts per second. See for more details the performance section in this document.
- Interrupt handlers can be very short leading to a more responsive system.
- Interrupts are allowed to use most of the RTOS functions including all signaling functions.

Many competing systems claim zero interrupt latency. While that can be correct only the segmented interrupt architecture allows those interrupts to communicate with the rest of the system. These interrupt handlers are standalone and can't be very functional. Those claims are more for marketing purposes than anything else because if an interrupt handler can signal events, release semaphore or write in queues they are use-less.

See the architecture diagram at the next page.

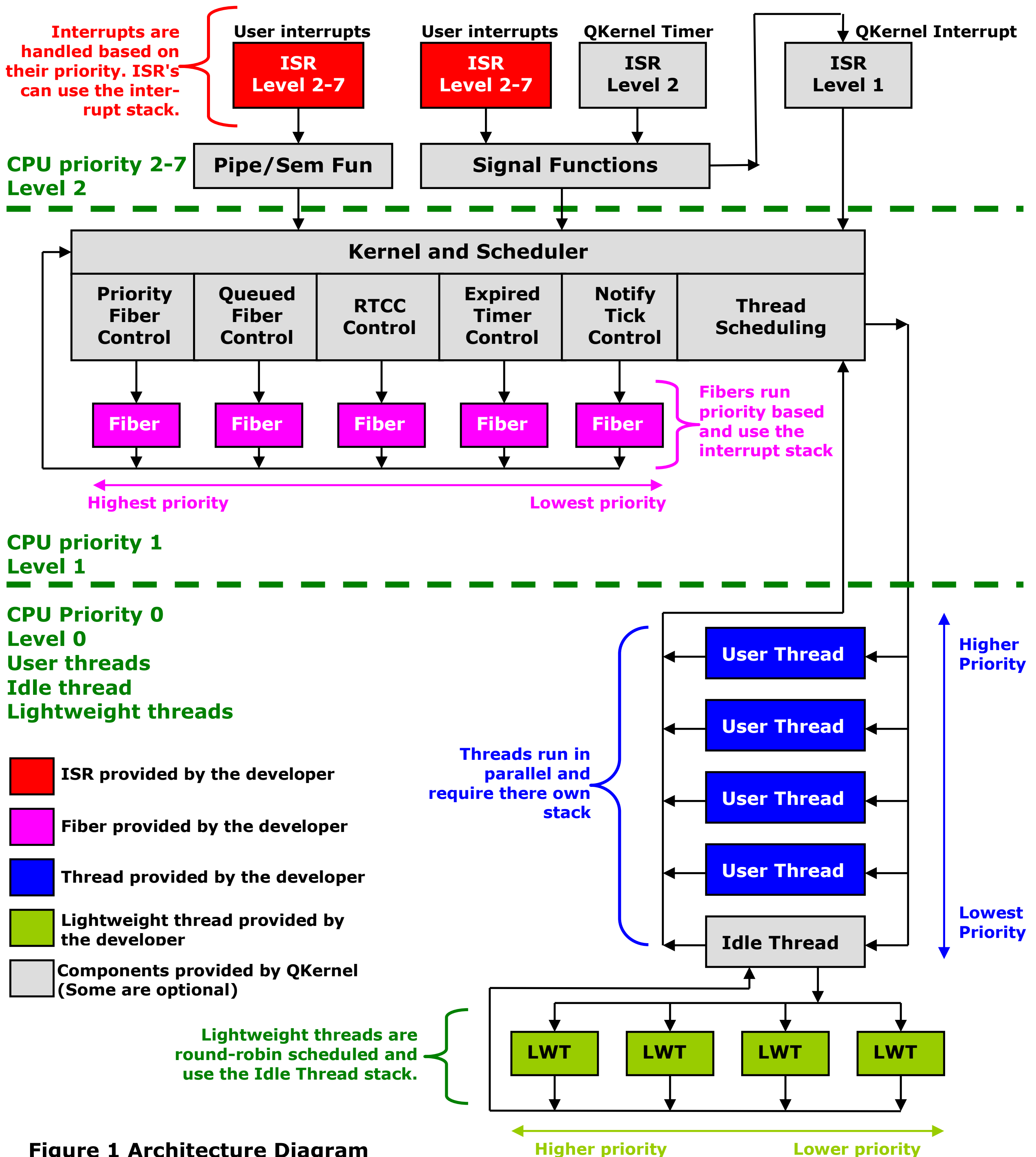


Figure 1 Architecture Diagram

This diagram indicates at which CPU level every components runs and which components are provided by ***Q·Kernel***. All threads and lightweight threads run at CPU priority 0 and can be interrupted by the Kernel/Scheduler which runs at CPU priority 1. The Kernel/Scheduler can be interrupted by ISR's. Signal functions always use the ***Q·Kernel*** interrupt for synchronization purposes. ISR's that use Pipe functions can directly communicate with the kernel.

2. Architectural features

2.1. Dual-Mode RTOS

A new breed of embedded applications is rapidly evolving. Traditional DSP applications are adding networking and other control functionality. But at the same time, the typical MCU control application will often include High Dataflow requirements like streaming media and other DSP functions. An emerging solution for this new class of 'hybrid' application is the convergent processor. This design approach combines both DSP and RISC/microcontroller capabilities into a single, unified architecture.

A processor that implements this architecture can operate as a DSP engine, be totally dedicated to a control application, or can operate somewhere in between. This makes those processors suitable for everything from industrial control to portable devices. Single convergent processors are an attractive alternative to the larger and more costly RISC and DSP processors. The dsPIC and PIC32MX from Microchip are convergent processors that can replace the RISC and DSP processors.

While Microchip implemented the hardware for convergent processing, the software is often lacking. A traditional multithreading RTOS adds enormous overhead to the DSP portion of the application. While a simple scheduler may work fine for the DSP or High Dataflow part of the application, it is not a good solution for a control application.

The traditional RTOS requires a stack for every thread, so it can block while waiting for an event and can be preempted by a higher priority thread. The switching between threads, known as context switching, is an expensive operation especially for processors with lots of context registers[1] like the dsPIC/PIC24 (20) and PIC32 (36). The larger the context, the longer it takes to switch the context.

DSP and High Dataflow applications typically read a block of data, operate an algorithm on the data, and then send the data to another programming unit for further processing. Due to the real time nature of the data, the algorithm must start within a very tight window once the data becomes available. Developers often design their own custom executive that handles the High Dataflow based on a corporative scheduling model. To combine this model with a traditional RTOS they run the algorithms in a high priority thread which adds a lot of overhead.

The Dual-Mode RTOS combines the traditional thread-based kernel architecture for real-time control processing with specialized fibers for High Dataflow operations. The architecture accommodates the different needs for both domains by separating them. ***Q·Kernel*** enables both types of application code to run fully optimized on a single processor, and both fibers and threads use a common API.

In order to meet real-time requirements, the DSP and High Dataflow processes run as fibers, at a priority that is higher than control threads which ensures they get access to the CPU. These fibers are lightweight because they have no context, making the switch from fiber to fiber very fast. Furthermore, fibers run at a priority just below that of interrupt handlers, a position that tends to reduce startup latency and minimize jitter.

---

[1] Context registers are not only the primary registers, but also the supporting registers like RCOUNT, SPLIM, TBLPAG, PSVPAG, etc. for the 16-bit PIC's and EPC, SR, HI, LO, etc. for the 32-bit PIC's.

2.2. Low Power and Tick-less

Many embedded applications spend most of their time waiting for an event to happen – a touch on a panel, incoming communication or wait for a time delay. In many applications the processor is only active for a small amount of time and battery life can be extended significantly by placing the processor into idle or sleep mode. Maximizing idle or sleep time, and minimizing active time is the key to extending battery life.

This means that state machines with looping and polling don't work well, but interrupt driven applications based on an RTOS do. While an RTOS is a much better solution, it has one disadvantage. Most RTOS require a tick for time management. This causes the processor to be frequently activated which consumes additional power. The shorter the tick-time, the more power is consumed. Applications often require a short tick-time for finer timing. A Tick-Less RTOS solves this problem.

Q·Kernel is tick-less and eliminates polling completely. It also optimizes power saving by splitting the timing into a human time scale (1 second to >30 years) and a processor time scale (1 µSec to 10 Sec). The human time scale uses the RTCC or the 32 KHz timer that is available in sleep mode and provides more power saving. The processor time scale provides a wait time with a granularity of 1 µSec. When there is an outstanding short time request, the processor can be switched to idle mode. If there is no outstanding short time request, the system will stop the timer and switch to sleep mode. This means that power consumption is always minimized while the system is waiting for user activity.

2.3. Integrated Power Management

Low power consumption is a deciding factor in many embedded applications. More and more applications like medical devices, wireless communications and personal devices demand low power consumption for a better battery life. Most hardware vendors, like Microchip, have equipped their processors with power saving features.

Most processors, including the 16 and 32 bit chips from Microchip, provide several power saving modes. ***Q·Kernel*** contains integrated power management that makes it simple to lower power consumption. When the RTOS is idle, it signals the application and provides the best power saving mode. The application has the ability to disable additional hardware and instruct the RTOS to select the required power mode. The power management module will select this mode without suffering from race conditions. ***Q·Kernel*** handles all race conditions and makes the implementation simple and flexible.

2.4. Never Disables Interrupts

Competitive systems are designed to disable interrupts during critical operations. ***Q·Kernel*** is designed to never disable interrupts. The "Segmented Interrupt" architecture makes losing interrupts a thing of the past. This unique architecture will never disable interrupts and the combination with fibers makes interrupt handling very fast. As a result ***Q·Kernel*** does not add a single cycle to the interrupt latency and facilitates seamless communication between Interrupt Service Routines, threads and fibers. It can handle very high interrupt rates. In the Thread metrics interrupt performance test an interrupt signals a waiting thread. A PIC24H running at 40 MHz can handle more than 570.000 of those tests per second.

The interrupt processing prevents Interrupt jitter. Reducing the jitter is important for Real Time Operating Systems, since they must maintain a guarantee that execution of specific code will complete within an agreed amount of time. Only "Segmented Interrupt" architectures are able to guarantee jitter free operation.

2.5. Zero Interrupt Latency and No Interrupt Jitter

Interrupt latency is the time between an interrupt request and the execution of the first instruction of the Interrupt Service Routine. The interrupt latency is the sum of a lot of different smaller delays explained below.

The first delay is typically in the hardware. Modern hardware architectures use instructions that are single cycle or double cycle. Hardware typically introduces 4 to 8 cycles of fixed interrupt latency.

The second delay is related to the RTOS and the fact that interrupts are disabled. Most RTOS are based on the "Unified Interrupt" architecture temporarily disabling the interrupts to protect critical sections including thread switching. This contributes to longer interrupt latency.

Q·Kernel has a segmented interrupt architecture that never disables interrupts, but will postpone communication with the RTOS when a critical section is detected. This approach solves the interrupt jitter and guarantees zero interrupt latency. Jitter is the variation in interrupt latency. **Disabling interrupts always leads to interrupt jitter.**

2.6. Hard Real Time

A hard real-time system, also called deterministic or temporal correct, is a system that requires a guaranteed response to specific events within a defined time period. The failure of a hard real-time system to meet these requirements typically results in a severe failure of the system.

The correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed. The system can be seen as a software extension of the hardware ISR mechanism and guarantees that the highest priority thread that can run shall run. To optimize the deterministic behavior of the system ***Q·Kernel*** implements the following mechanisms:

- Zero Interrupt Latency
- Interrupts are never disabled
- Memory allocation and de-allocation is deterministic

This approach delegates the timing of the application ***Q·Kernel***, so the developer can focus on the behavior of the system.

A Hard Real Time RTOS like ***Q·Kernel*** supports the developer to create a dependable and stable system. If the application is tested it remains working since the timing will always be the same.

3. Feature and functionality

Q·Kernel provides the basic functionality that most competitors provide and more advanced features like a deterministic variable memory system, managed messages, fibers and lightweight threads, statistics and tracking and an advanced publish/subscribe mechanism that promotes re-use of code and development agility.

3.1. Threads, Fibers and Lightweight Threads

Q·Kernel combines traditional thread-based kernel architecture for real-time control processing, specialized fibers for High Dataflow operations, and Lightweight Threads for efficient cooperative multi-threading.

In order to meet real-time requirements, High Dataflow processes run as fibers, at a priority that is higher than control threads which ensures they get access to the CPU. These fibers are lightweight because they have no context, making the switch from fiber to fiber very fast. Furthermore, fibers run at a priority just below that of interrupt handlers, a position that tends to reduce startup latency and minimize jitter.

Lightweight threads are executed when all threads are in a wait state and no interrupts or fibers are running. Lightweight threads and regular threads share the same stack, which limits RAM requirements and makes them ideal for low priority, non-time critical drivers (displaying text on an LCD or logging data in flash). Competitive products use threads for all drivers but applications that use ***Q·Kernel*** can significantly limit resources by using lightweight threads.

3.2. Memory management

Most Real Time Operating Systems provide the developer with a simple fixed-size-blocks memory allocation algorithm. This works very well but is not very flexible because the developer needs to know the memory requirements in advance, including size. ***Q·Kernel*** support three memory allocation mechanisms so the developer can always use the optimal mechanism. The memory allocation mechanisms are not only versatile but also much faster than any competitor.

- **"Allocate only" heap**. In embedded systems, many blocks are permanently allocated at startup. The heap works well because each block can be exactly the right size. Fragmentation is not a problem because these blocks are never released. The used algorithm is fast and deterministic.
- **Fixed Memory Blocks. *Q·Kernel*** also implements the traditional fixed-size-blocks memory allocation algorithm but added the extra facility to allocate and de-allocate from interrupt handlers. The used algorithm is fast and deterministic.
- **Variable Memory Blocks.** The Variable Memory Block algorithm manages blocks of every size and functions like malloc() and free(). Allocation is based on the required size of the memory block. The algorithm is fast and reasonable deterministic and there is no external fragmentation. Allocation can also be based on the pool and in that case the allocation and free are 100% deterministic and extremely fast.
- **"Allocate only" EDS heap.** Some of the devices in the 16-bit PIC line support Extended Data Space memory management and this makes it

possible to address many megabytes of data space. To address this memory the system uses 32-bit pointers and the DSWPAG and DSRPAG registers. The EDS heap works exactly the same as the standard heap but memory is it returns.

- **Variable EDS Memory Blocks.** Variable EDS Memory blocks work the same as normal Variable Memory Blocks as describe above but the system returns 32-bit pointers.

Q·Kernel is the only vendor that specifies exactly how many cycles it takes to allocate and de-allocate memory.

3.2.1 Extended Data Space for Thread Stacks

Some of the devices in the 16-bit PIC line support Extended Data Space memory management and this makes it possible to address many megabytes of data space. Threads can have their stack in EDS while they still operate at full speed without wait states. This also means that the developer can create many threads. Switching to a 32 bit processor is not always required.

While context switching from a thread with an EDS stack to another thread with an EDS stack take some more time switching from an EDS stack to a thread with a standard stack is just as fast as there were no EDS stack threads. See also the Performance chapter in this manual.

3.3. Statistics and Thread & Fiber Tracking

Q·Kernel provides detailed statistics per thread and system overhead. It provides a very accurate picture how much CPU time is used per thread, by ***Q·Kernel*** itself and how much free CPU time there is.

While statistics are very use-full they change the timing of the application. Fiber and Thread tracking is non-intrusive and does not change the timing. Tracking makes it possible to follow the application timing in detail with a logic analyzer including individual threads, priority and queued fibers and the ***Q·Kernel*** scheduler.

Another form of tracking is the switch notification. A user function will be called when the system switches threads. The developer can write code to solve difficult to find issues or can code extensive tracking mechanisms for debugging purposes.

3.4. Real-time Clock, Alarm and Timer functions

Q·Kernel provides a software real time clock for human timing requirements and timer functions for processor scale timing. Because ***Q·Kernel*** is tick-less it provides timing functions with the granularity of 1 µSecond up to 30 years. The RTCC or a 32 KHz crystal with TMR1 can be used to create the software real-time clock or it can be emulated from the processor clock.

Both the RTCC and processor clock are completely disconnected from threads which makes it very versatile. Expired timers or RTCC alarms run as fibers and use the interrupt stack. Because timers are managed objects they can be created, deleted and opened. Re-use of timers is possible.

Some competing products set event flags. This results almost always a context switch, because a thread is waiting for that event. In a lot of cases this is not necessary because all the work can be done without a context switched.

Q·Kernel also provides integration with the timing functions of the TCP/IP stack for easy integration.

3.5. Centralized Error Handling

Centralized Error handling minimizes code complexity. When a ***Q·Kernel*** function returns, the developer knows that the function ended successful so it is not necessary to test the result. When an error occurs control resumes at the central error handler instead of being passed back to the caller.

The central error handling is very efficient during debugging because the developer only has to set one breakpoint for all fatal errors. All errors have a unique code so the developer knows immediately what's wrong. The system can also be used for application errors.

Errors can occur in interrupts and in other places where it is not possible to log them directly and it could be that the system is not stable anymore because of the error. ***Q·Kernel*** has a special feature that makes it possible to log the errors after a reset.

Q·Kernel is one of the few products that use the stack overflow detection of the processor if available. This means that undetected stack overflow is a thing of the past and provides the developer with early detection of problems. This makes the developer more efficient. Stack overflow and other exceptions are integrated in the centralized error handling.

3.6. Scheduling

Q·Kernel supports preemptive and cooperative scheduling. Preemptive scheduling involves the use of an interrupt mechanism which suspends the currently executing thread and invokes the scheduler to determine which thread should execute next.

Interrupts or the scheduler can preempt a thread or a thread can preempt itself and this always happens immediately. There is no time-delay between the moment that a thread with the highest priority wants to run and the moment the context switch will be enabled.

Cooperative scheduling occurs when the programmer yields execution to another thread programmatically. Cooperative multi-threading can be implemented with threads, but lightweight threads are a better solution for cooperative multi-threading. They use fewer resources than threads.

3.7. Minimize RAM usage

Q·Kernel is designed to minimize RAM usage to make as much as possible RAM available to the application. The most important RAM saving comes from the interrupt stack, fibers, lightweight threads, Shared stacks and dynamic memory allocation.

- The interrupt stack can be used by all interrupt handlers, fibers and lightweight threads. Interrupts occur at unpredictable times so all threads have to accommodate the worst case interrupt memory usage which lead to large thread stacks and a waste of memory.
- Fibers use the interrupt stack and can minimize the number of threads. This can save a lot of memory because thread stack space is not used.

- Lightweight threads also use the interrupt stack and can minimize the number of threads. This can save a lot of memory because thread stack space is not used.
- Shared Stacks can be used to limit memory allocation for some PIC24F and all PIC24E and dsPIC33E devices. This method uses shared stack space in the low memory range (0-32k) and dynamically exchanges this with upper memory (64k). Threads with shared stacks run with exactly the same speed as standard threads but save lots of lower memory.
- Almost all memory requirements, including stacks, messages, objects and application memory can be freed if not used anymore. Almost all services have a "Create" and a "Close" function including threads. This means that drivers that run in threads don't require memory if they are not used.

Functional applications can be built on a MCU with only 2kb of RAM.

3.8. Advanced API

Strong naming conventions are imposed on the API for optimal code readability and to make function-names unique. All function names are based on polish notation and names always start with a lower case **q** to make then unique.

The functions are named in qSrvFunction() where Srv is the service like Evt, Sem or Thr and Function is the name of the function like Create, Open, etc. Combined examples are qEvtCreate(), qEvtOpen(), qSemCreate(), qThrCreate(), qThrOpen().

Q·Kernel tries to keep the number of parameters as low as possible. Most function only has one or two parameters for optimal code readability. All wait functions like qThrOpen(), qEvtWait() qMsgReceive() etc. are provided in three variants, the normal version that never times out like qMsgReceive(), a non-blocking version like qMsgReceiveNB() and qMsgReceiveTO() where the developer can specify the time-out.

3.8.1 Type save parameters

Q·Kernel provides the developer with advanced API that accepts type save parameters so errors are found during compile time instead of run-time. Every kernel object has its own type like qtQUE for a queue object or qtEVT for EventSet objects. Mixing the types will be detected at compile time. Type save parameters are checked at compile time so they don't have to be checked at run-time making it faster.

3.9. Threads

A thread contains the code to perform an application task. While tasks can also be performed by fibers or lightweight treads the main differences is that threads can wait on an event, wait for a mutex to become unlocked, wait for a semaphore etc. The number of threads is only limited by the amount of available RAM.

A ***Q·Kernel*** based solution is multi-threaded meaning that every thread operates on its own. It can shield all functionality from other threads or can communicate with other threads.

Threads can be in a 3 states:

- Run State, meaning that the thread is currently running. Only one thread in the whole system can be in that state.

- Ready State, meaning the thread is ready to run but the system does not allow it to run because there are threads with a higher priority that are also in the ready or run state.
- Wait State, meaning that the thread is waiting on something to happen. That can be an event a timer to expire, etc.

Threads have their own EventSet they can wait for. Other threads, fibers or lightweight threads can signal the EventSet but can't wait for it.

Threads can be created or removed by other threads. The thread object is managed object that can be created, opened or closed. A thread that has done its tasks and ends it execution is automatically removed by the system and its resources are returned to the resource pool.

3.10. Fibers

Q·Kernel implements threads, fibers and lightweight treads. Threads are normally used for control tasks and fibers are mostly used for high dataflow tasks. Fibers are optimized for cooperative scheduling and very fast interrupt response to support the tight time window for high speed dataflow applications. Fibers operate in a single stack environment to sustain the required very fast context switches.

Q·Kernel provides three types of fibers, priority based fibers, queued based fibers and scheduled fibers.

- Priority based fibers run directly after interrupts and is the fastest way for an interrupt to communicate with the rest of the system. The overhead is extremely low and the interrupt handling is very fast.
- Queue based fibers run after the priority fibers and as the name implies are queued together with one or two parameters. These fibers are mostly used to implement drivers.
- Scheduled fibers are functions that are started by the scheduler and are expired timers, alarms etc.

The use of fibers saves precious memory because they use a single stack mechanism. The priority and queued based fibers give ***Q·Kernel*** a significant advantage over the competition for fast interrupt and data handling. The context switch time is also significantly faster.

3.11. Lightweight Treads

The main purpose of lightweight threads is to provide a very efficient cooperative multi-threading mechanism and minimize memory requirements. Lightweight threads are executed based on their priority if all threads are in a wait state and there are no interrupts or fibers running. They look a lot like fibers but they run at a low priority in the idle thread.

Lightweight threads are an ideal candidate for low priority, non-time critical drivers. Every application has a number of those, like displaying text on an LCD, logging information in flash, etc. Competitive products use threads for all drivers but applications that use ***Q·Kernel*** can significantly limit resources by using lightweight threads.

As mentioned above lightweight threads look a lot like fibers meaning that all function must be called without blocking with the exception of a sleep function. While fibers are always stateless, lightweight threads contain a limited amount of state. Lightweight treads can wait can yield to have a very simple facility to wait a specific time before called again. This unique feature makes lightweight thread simple to use.

4. Managed Kernel Objects

Kernel object are used by threads, fibers and lightweight threads to communicate with mutexes, semaphores, timers etc. They will be created by one thread and can be opened by other threads.

Competitive products use a global variable so multiple threads can use the object. Other threads don't know if the object is already created and it requires the developer to build code to manage that.

Q·Kernel objects are managed and a thread that opens an object can wait until another thread creates it. The name of the object is used as an identifier for the object. This mechanism simplifies the synchronization during system startup.

The managed kernel objects all have three function calls in common. The "Create" to create a new object, "Open" to open an existing object and "Close" to end the use of that object and return the resources like memory back to the pool.

4.1. Mutexes

A mutex object is a synchronization object to provide threads access to shared resources. Only one thread at a time can own a mutex object, whose name comes from the fact that it is useful in coordinating mutually exclusive access to a shared resource. A mutex is a managed object so there is a qMtxCreate(), qMtxOpen() and qMtxClose().

Any thread with a handle to a mutex object can use the qMtxLock() function to request ownership of the mutex object. If the mutex object is owned by another thread, the wait function blocks the requesting thread until the owning thread releases the mutex object using the qMtxUnlock() function. The return value of the wait function indicates whether the function returned with the state of the mutex being locked or it timed-out. If more than one thread is waiting on a mutex, the thread with the highest priority is selected.

After a thread obtains ownership of a mutex, it can specify the same mutex in repeated calls to the wait-functions without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. The thread only has to call the qMtxUnlock() once to release ownership of the mutex.

In scheduling, priority inversion is the scenario where a low priority thread holds a shared resource that is required by a high priority thread. This causes the execution of the high priority thread to be blocked until the low priority thread has released the resource, effectively "inverting" the relative priorities of the two threads. If some other medium priority thread attempts to run in the interim, it will take precedence over both the low priority thread and the high priority thread. ***Q·Kernel*** implements the priority inheritance algorithm to eliminate priority inversions.

4.1.1 Critical Sections

Q·Kernel defines a critical section as a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread, lightweight thread, fiber or the scheduler. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use.

The ***Q·Kernel*** implementation of a critical section suspended thread switching. This is a very effective low cost way of implementing a critical section. It takes two cycles to disable the thread switch and it will take also take two cycles to resume thread switching. If a thread switch request was queued it will be executed immediately after the critical section ended. Simple defines are available to implement this behavior.

4.2. Semaphores

Conceptually, a semaphore maintains a set of permits. Semaphores can be acquired and released. 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.

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource or protect access to a resource that contains multiple entities.

A thread or fiber uses the qSemCreate() function to create a semaphore object. The creator specifies the initial number of permits and a name for the semaphore object. Threads can open an existing semaphore object or wait until the semaphore is created by specifying its name in a call to qSemOpen() function.

When a thread acquires a permit, the count is decremented and when a thread releases a permit the count is incremented. When a thread attempts to acquire a permit and no permit is available the thread will be preempted. If more than one thread is waiting on a semaphore, the thread with the highest priority is selected.

Releasing an permit is possible from an interrupt, fiber or thread.

4.3. EventSets

Threads and fibers can use EventSet objects in a number of situations to notify a waiting thread about the occurrence of mix of events. EventSets are groups of 16 or 32 binary flags, called event flags, that describe conditions. Threads can wait for those flags (conditions) to be set. For example, a thread waits for any of 6 conditions when it has to close a valve. Other threads or fibers can set one or more conditions.

Q·Kernel implements two type of EventSets, the Thread EventSets and Global EventSets. Thread EventSets belong to the thread and only that thread can wait on the EventSet, while multiple threads can wait on Global EventSets. Thread EventSets are faster and require less overhead.

Multiple threads can wait on ANY combination of events flags in one EventSet or on ALL event flags in one EventSet.

When a thread signals an EventSet, any number of waiting threads that specify the same EventSet in one of the wait functions, can be released. If more than one thread is released, the thread with the highest priority is selected to run.

The implemented signaling algorithm allows multiple threads to wait with the clear option. This is a significant difference with competing products because they clear flags during the signaling process and that makes signaling unpredictable for other threads or limits the functionality. Signaling an EventSet is possible from an interrupt, fiber or thread. This contributes to the flexibility of the EventSets.

4.4. Pipes

Pipes allow communication between threads, fibers and ISR's and are designed to support high speed communication without a lot of thread switching.

Fast ISR's normally operate in the 1 to 10 µSec time frame. A thread operates more in the 1 millisecond time frame, hundreds of times slower than an ISR. Pipes are designed to connect the two time domains by providing a FIFO buffer for information storage and exchange.

Q·Kernel pipes use a FIFO buffer that concurrently can be written to and read from by threads, fibers, lightweight threads and Interrupt Service Routines.

Pipes are not just FIFO buffers that allow concurrent reading and writing but they also must have the ability to block a thread that tries to write to a full pipe or tries to read from an empty pipe. Also a blocked reading thread must be activated when there is (enough) information in the pipe and a blocked writing thread must be activated when the pipe has (enough) space. ***Q·Kernel*** provides a synchronization mechanism that is very flexible so pipes can be used in a lot of different situation. Sometimes a reading thread must be activated as soon as one element is put in the pipe. In other case the reading thread must be activated when more than 80% of the pipe is full to minimize context switches. This kind of flexibility can be best realized by notification functions. Those notification functions are defined during the creation of a pipe and are called "Notify Reader" and "Notify Writer". When something is written in a pipe the reader notification is called and when something is read from a pipe the writer notification is called. The notification functions can take any action to synchronize threads or just do nothing and wait until there is more information in the pipe. The notification functions are called with two parameters; a pointer to the pipe object and the number of blocks that are written into the pipe or read from the pipe. This information can be used to handle device information.

As describe above the flexible signal functions allows the developer to combine the hardware queues with pipes and still find the optimum signaling moment.

4.5. Messages

The ***Q·Kernel*** implementation allows variable size messages and ***Q·Kernel*** keeps a use count. The use count keeps track how many threads are using the message. A message is allocated by the qMsgAlloc() function and the message count is 1. If a message is send the use count is incremented because the message is in transport. If a thread de-allocates the message with the function qMsgFree(), the use-count is decremented. Both sending and receiving thread need to de-allocate the message before the use-count reaches 0 and the object is returned to the memory pool. Every thread handles the message as if it owns it. It does not have to administrate this behavior itself. Message can also be allocated from interrupts. This functionality combined with pipes makes allocating, de-allocating and sending and receiving from interrupts a reality. Messages can be sent by using queues, pipes or can be used in the publish/subscribe mechanism.

4.5.1 Queues

Queues are the primary means of sending and receiving messages between threads, fibers or lightweight threads. Multiple messages can reside in a queue. Queues are very fast because they exchange pointers between threads and a

sending thread will recognize that there is a receiver and will short-cut the transfer. Competitive products work differently and are much slower.

The function qQueReceive() reads a message from the queue and wait if there is no message available. The function qQueSend() is used to send a message to the queue and waits if the queue is full.

Often it's required to synchronize waiting for messages with other events or to send messages from interrupts. This functionality is provided with pipes. Messages can be sent and received from pipes, which provide more flexiblity.

4.6. Publish/subscribe functionality

Publish/subscribe (or pub/sub) is a messaging pattern where senders (publishers) are unaware of specific receivers (subscribers). Subscribers express interest in one or more messages, and only receive messages that are of interest, without knowledge of what, if any, publishers there are.

This decoupling of publishers and subscribers is called loosely coupling and has a number of advantages:

- A change in one module does not force a ripple-effect of changes in other modules. Developing and maintaining software requires less effort and time due to the decreased inter-module dependency.
- It will be easier to reuse software because there are fewer dependencies. A software module will be easier to test because dependent modules do not need to be included.
- No changes are required if the number of subscribers or publishers changes.
- This pattern promotes agility because a change in the application does not require that all software modules have to be changed and re-tested.

The ***Q·Kernel*** implementation of pub/sub is very simple to implement. A publisher creates a publish object with the function qPubCreate(). Every publish object has a name and other publishers and subscribers can get access to this publication by opening the object by name.

Subscribers can subscribe to a publication and specify the message delivery method.

The message can be delivered by calling a delivery function, by sending it to a queue or writing it into a pipe. Depending on the required delivery method the subscriber has to use one of the following functions, qPubSubscribeFun(), qPubSubscribePip() and qPubSubscribeQue(), to subscribe to a publication.

- **Function delivery** can be used to decouple the time domains from publisher and subscriber.
- **Pipe delivery** can be used when no information can be lost and the developer needs full control over the information flow.
- **Queue delivery** can be used when no information can be lost but the load is limited. This is the most simple delivery method to implement.

Messages are published with the function qMsgPublish() and the payload is a ***Q·Kernel*** message (qtMSG). The ***Q·Kernel*** message infrastructure helps the

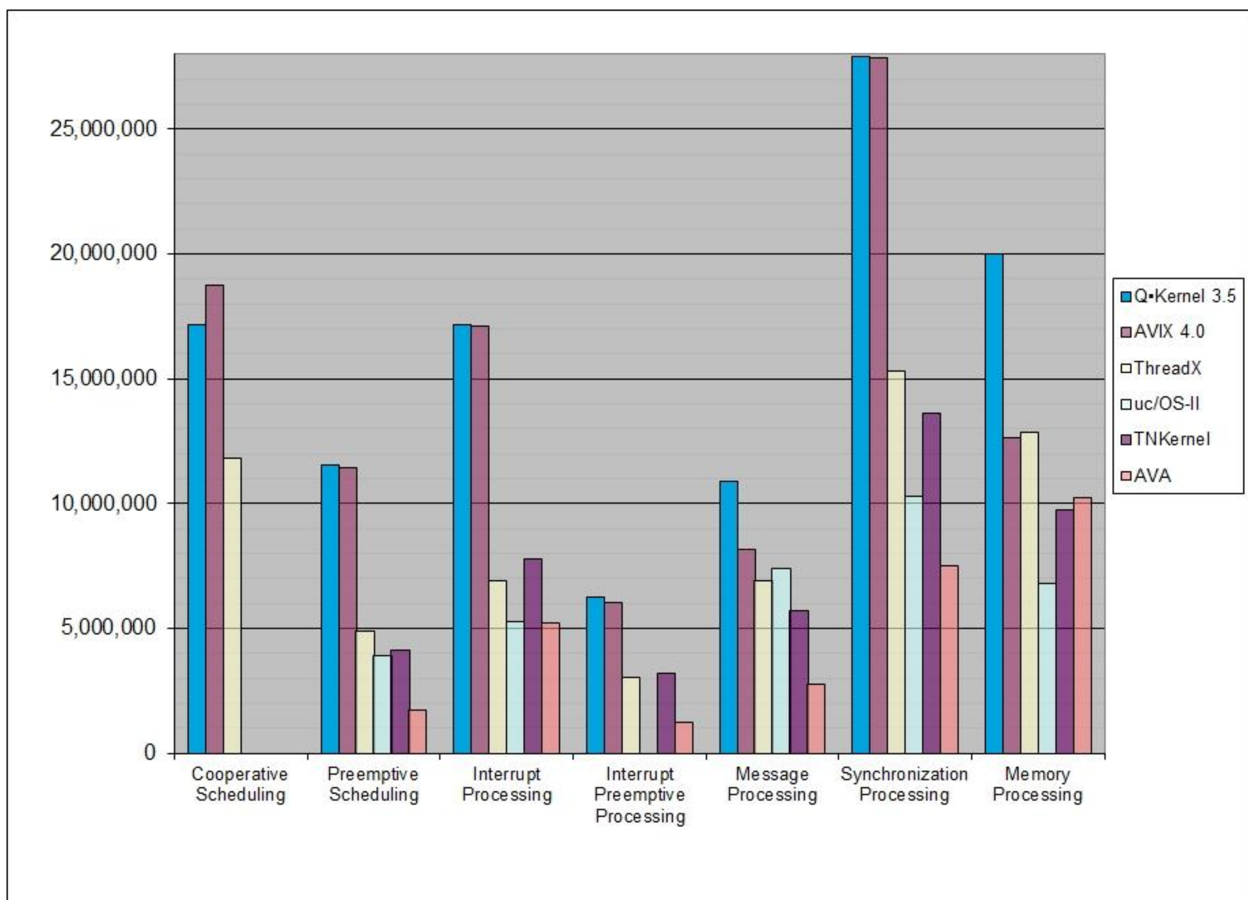
developer with the life-time of the message. Every individual subscriber just has to free the message when done and ***Q·Kernel*** will manage the life-time.

5. Performance

Q▪Kernel delivers the best overall performance of all competitors according the Thread-Metric benchmarks test suite produced by Express-Logic Inc.

Q▪Kernel is one of the best performing RTOS's, because it is tick-less and it uses the segmented interrupt architecture. The results[2] are in the following table where the yellow cell specifies the best performance.

	Cooperative Scheduling	Preemptive Scheduling	Interrupt Processing	Interrupt Preemptive Processing	Message Processing	Synchronization Processing	Memory Processing
***Q▪Kernel* 4.0**	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



[2] 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.

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[3]**
AVIX	8505
Q·Kernel with use of fast[4] library	7418 (14% smaller than AVIX)
Q·Kernel with use of standard[5] library	6996 (18% smaller than AVIX)
Q·Kernel with use of small[6] 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.

5.1. Tick-less operation

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.

---

[3] This is the size in words and compiled with version 3.25 of the C30.

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

[5] The standard library provides complete and constant error checking but creates a larger footprint.

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

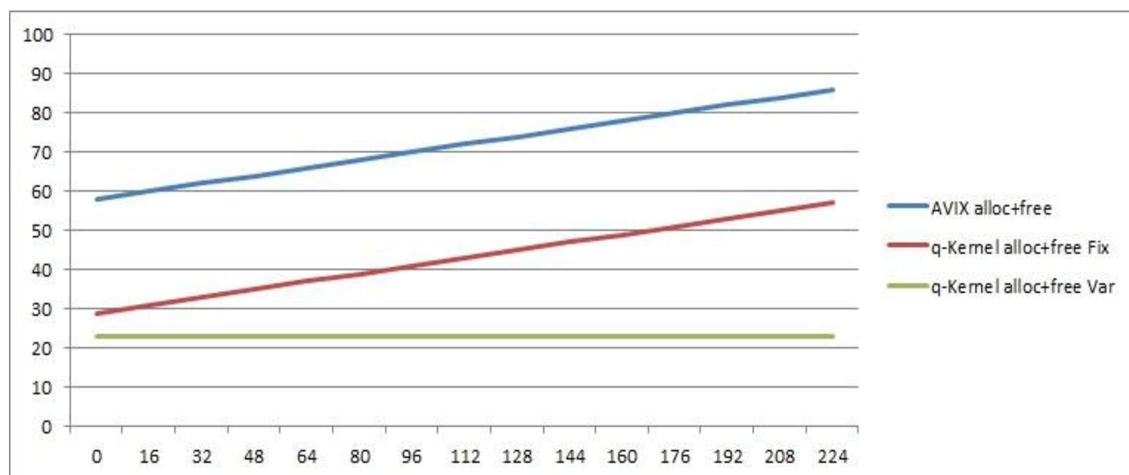
5.2. Memory allocation

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 and 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.

5.3. Context switching

Context switching is extremely fast. ***Q·Kernel*** uses a different mechanism for preemptive context switches en non-preemptive context switches[7]. Threads that have their stack in EDS memory switch a bit slower because they have to copy their stack. This is often not a problem. The information below specifies the context switch time in cycles base on a 100 byte Stack for EDS switches

Context switch time in cycles

Type of context switch	Non Pre-emptive	Pre-emptive
Fully optimized no EDS stacks	40	60
Standard Stack Thread → Standard Stack Thread	46	66
Standard Stack Thread --> EDS Stack Thread (cached[8])	46	66
Standard Stack Thread --> EDS Stack Thread (not-cached)	263	283
EDS Stack Thread --> Standard Stack Thread	46	66
EDS Stack Thread --> EDS Stack Thread	263	283

Based on a 70MIPS PIC24E the fastest switching time is about 0.6 µSec and the slowest is about 4 µSec.

Even the slower PIC 24F running at 16MIPS switches at 2.5 µSec.

[7] Non-Preemptive context switches are context switches that are not introduced by interrupts. Examples are waiting for a mutex, waiting for a events etc.

[8] Cached means that a Standard Stack Thread switches to an EDS Stack Thread and that EDS Stack Thread was still in memory. This happens in practice quit often if an EDS Stack Thread is preempted by a Standard Thread stack and if that thread is done it switches back to the EDS Stack Thread.