Q▪Kernel™

User Guide

Version 6.0-3353

Q▪Kernel™ is a product of QuasarSoft Ltd.

License

Q-KernelFree 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-KernelFree 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 Avenue Suite No. 354

Cochrane Alberta T4C 2E3

Canada

Tel. +1 (403) 450 3482

www.quasarsoft.com

About this Document

This document assumes that you already have background knowledge of the following:

- The software tools used for building your application, mainly the compiler and linker
- The C Programming language
- The processor

If you feel that your knowledge of C is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, covers the ANSI C standard.

The Q▪Kernel™ Reference Guide is available to learn the API.

How to Use this Manual

The intention of this manual is to give you a detailed introduction to Q▪Kernel™. See the following scenarios:

- Non-experienced RTOS developers should read chapters 3-8 to learn the basics of an RTOS.
- Experienced RTOS developer should read chapters 3, 4, 6, 11 and 12 to learn the differences between new generation RTOS like Q▪Kernel™ and existing RTOSes.
- Experienced Dual-Mode RTOS developer should read chapter 22 to get a fast start.
- Unique Q▪Kernel™ functionality is described in chapter 5, 7, 10, 11, 19 and 20.

This document also serves as a source of information about Q▪Kernel™ and reveals a lot of information about the internals of the system.

1. Introduction to Q▪Kernel™

Q▪Kernel™ is a Tick-Less Dual-Mode Real Time Operating System (RTOS) sometimes referred to as a kernel. Q▪Kernel™ is specially created for the modern processors and fully exploits the power of the processor and the development environment.

1.1. What Makes Q▪Kernel™ Unique?

Q▪Kernel™ has some unique features not found in any other RTOS. The list of features is very complete, but a number of features really separate Q▪Kernel™ from the competition. Some of the features are describe below.

1.1.1 Dual-Mode RTOS

Q▪Kernel™ combines the traditional thread-based kernel architecture for real-time control processing with specialized fibers for DSP and 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.

1.1.2 Low-Power Operation

Q▪Kernel™ provides idle detection and can switch the processor in a low-power mode on demand. This feature combined with Tick-Less operation provides the developer with the best tool to create applications that consume minimal power.

1.1.3 Tick-Less Operation

Q▪Kernel™ operates Tick-Less that provides true µSec wait-time granularity and decreases power consumption significantly.

1.1.4 Zero Interrupt Latency

Q▪Kernel™ never disables interrupts, not for a single cycle, and provides true Zero Interrupt Latency and eliminates interrupt jitter.

1.1.5 Advanced Interrupt Stack

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

1.1.6 Advanced Memory System

Q▪Kernel™ supplies the developer with an advanced memory system, called variable memory blocks. This makes designing systems simpler and limits the RAM footprint.

1.1.7 Dynamic System

Q▪Kernel™ is a dynamic system, meaning that resources can be returned to the resource pool when they are no longer required. This modern approach makes the design simpler, more functional, and limits the RAM footprint.

1.2. Why Use a Multi-Threading RTOS?

Multi-threading allows you to better utilize CPU resources. Multi-threaded systems are event driven and events can be handled as they occur, instead of looking for them and processing them as they are found. If one event has higher priority than another (e.g., a pump failure signal), Q·Kernel™ can immediately run the higher priority thread regardless of what was being done, and get back to the lower priority ones when it is finished with the higher priority thread. If polling had been employed and a lower priority function was just invoked before the higher priority one, it would go unrecognized until polled (programmatically) in the code or until the lower priority code is complete

Applications that don't use a multi-threading RTOS must use the "superloop" approach. Essentially, this is a program that runs in an endless loop, calling functions to execute the appropriate operations. No real-time kernel is used, so Interrupt Service Routines (ISR) must be used for real-time parts of the software or critical operations (interrupt level). This type of system is typically used in small and simple systems.

The "superloop" approach can become difficult to maintain if the program becomes more complex. Because one software component cannot be interrupted by another component (only by ISR's), the reaction time of one component depends on the execution time of all other components in the system. Deterministic behavior is therefore poor.

1.3. Why Q·Kernel™?

Q▪Kernel™ is built from the ground up based on a modern architecture, and designed to handle interrupts better than any other RTOS on the market. It is specifically written for microcontrollers, so it is small and very fast. It uses advanced algorithms to optimize speed and is very versatile. If you currently use another RTOS and experience memory limitation, bad real-time behavior, long interrupt latencies, or slow throughput, Q▪Kernel™ can help you solve these problems. Q▪Kernel™ significantly reduces the memory foot print compared to other systems and the hard real-time features of Q▪Kernel™ solve a lot of deterministic issues.

Q·Kernel™

The only free RTOS in the world with:

Dual-Mode and Tick-Less

Lowest Power Consumption

Integrated Power Management

Zero Interrupt Latency

Never Disable Interrupts

No Interrupt Jitter

Best Performance

Threads and Fibers

2. Q▪Kernel™ Benefits

Q▪Kernel™ is designed for embedded applications including consumer and industrial electronics like set-top boxes, measuring devices, and other portable and handheld devices. The kernel automatically scales to the right size.

As a customer, you will see the following benefits of using Q▪Kernel™ over other operating systems:

2.1.1 Dual-Mode RTOS

Q▪Kernel™ combines the traditional thread-based kernel architecture for real-time control processing with specialized fibers for DSP and 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.

This architecture allows the designer to use a less powerful chip and/or bring down the operational frequency to save power. A Dual-Mode RTOS also supports fibers. Fibers can save a lot of RAM and allows the developer to support the same functionality on a smaller (and cheaper) micro-processor.

2.1.2 License

Q▪Kernel™ is free software licensed under a modified GNU General Public License (version 3) as published by the Free Software Foundation. See the full license text at the following link <http://www.quasarsoft.com/license.html>

An alternative commercial license is also available in case that:

- You cannot fulfill the requirements stated in the "Modified GNU General Public License (version 3)"
- You require direct technical support and you which to have assistance with your development
- You require IP infringement protection

Q▪Kernel-Pro™ is a commercial licensed version of Q▪Kernel-Free™. This license does not contain any reference to the "Modified GNU General Public License (version 3)".

2.1.3 Royalty free

Both Q▪Kernel-Pro™ and Q▪Kernel-Free™ are royalty free.

2.1.4 Website

All documentation and programming code can be downloaded from the website. This means that communication is fast and the customer can always access the latest documentation.

2.1.5 Higher Quality

Higher quality is achieved by modular, well written code, automated tests, more than 96% code coverage, backward compatibility, extensive and usable documentation and a beta test program.

2.1.6 Faster Delivery

The high quality of the system and the unique Dual-Mode capability allows you to deliver faster.

2.1.7 Lower Maintenance

A staff of dedicated professionals maintains all our products and bugs are repaired quickly. Electronic distribution of the system provides fast access to the latest software.

2.1.8 More Functionality

Q▪Kernel™ provides much more functionality than competing products. This limits the amount of code that needs to be written by the developer and limits costs.

2.1.9 Best Error Handling in the Business

While we all try to prevent errors most of us know that debugging is a part of development. Q▪Kernel™ error handling is the best in the business and will help the developer find bugs faster.

2.1.10 Easy to Use

Q▪Kernel™ uses a consistent, intuitive API. No cryptic abbreviations and we provide complete documentation. This ensures that developers are productive sooner.

2.1.11 Support for the Most Advanced Interrupt Structure

The advanced interrupt structure of new microcontrollers is completely supported and allows the developer to build hard real-time applications. Q▪Kernel™ hides al complexities of the advanced interrupt structure from the developer, which increases productivity.

2.1.12 Separation of Concerns

Q▪Kernel™ enables the developer to concentrate on the specific application requirements, without having to worry about the timing and interrupt behavior. The deterministic and application aspects are separated.

2.1.13 Write Once and Re-Use

The unique structure of Q▪Kernel™ makes it easy to organize software in drivers. Those drivers are written, tested and documented once and can be re-used in every project saving you money and decrease the time-to-market.

2.1.14 Designed for Deterministic, Real-time Response

Real-time applications are called real-time for a reason. Real-time execution requires deterministic, fast interrupt response and fast context switching. Q▪Kernel™ utilizes an interrupt architecture that eliminates disabling of interrupts.

2.1.15 Maximize Development

Increase quality and eliminate embarrassing recalls by finding bugs early. Q▪Kernel™ memory management, stack limiting features and strongly typed parameters protect the developer from a wide variety of problems and maximize development.

2.1.16 Stack Tracing

Q-Kernel™ is one of the few Real-time Operating Systems that utilizes the native stack limiting features of the micro controller during all operating modes like interrupt handling, fibers, scheduler and threads. This makes it simple to find errors and minimizes development time.

3. Q▪Kernel™ Architecture

The Q▪Kernel™ architecture is based on modern operating system concepts like micro-kernel and "Segmented Interrupt" architectures. Q▪Kernel™ is by design a Dual-Mode RTOS to support modern hardware concepts.

With the unique capabilities of Q▪Kernel™, developers can optimize both the dataflow and control characteristics of the processor family. Such optimization ensures the efficiencies of applications running on these new generation devices that greatly reduce hardware cost and power consumption.

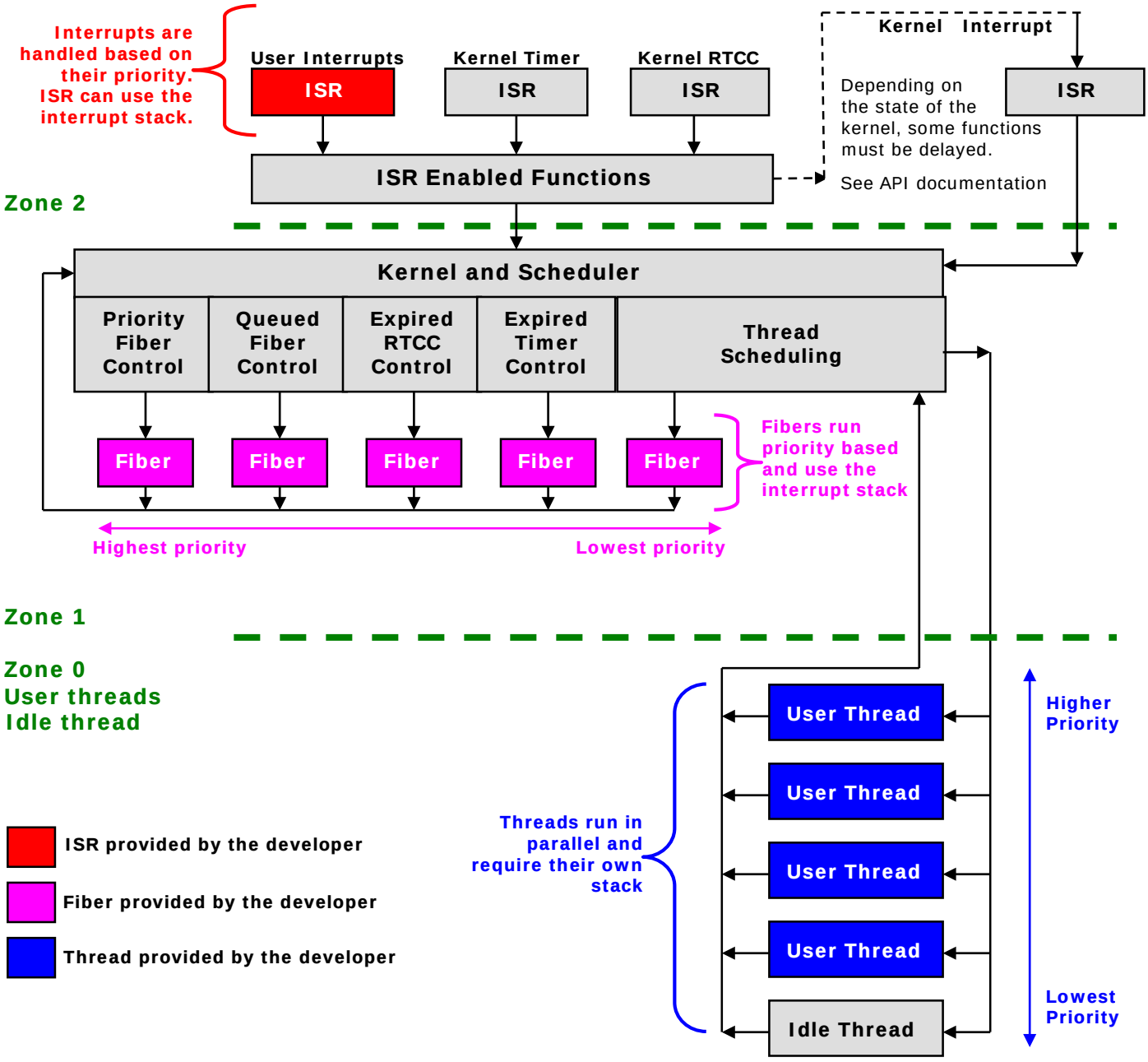
By combining a lightweight kernel, minimal context for high dataflow (like DSP), with a prioritized, preemptive kernel for event-driven threads, Q▪Kernel™ ensures that both control and high data load application code execute with maximum efficiency. Q▪Kernel™ is organized by priority so dataflow processing controlled by a dedicated scheduler always operating at a higher priority than control operations.

To implement this unique architecture the micro controller has to support 3 zones of operation, where a higher zone can interrupt a lower zone. The following page depicts the system design.

- Zone 2-15 contains all Interrupt Service Routines (ISR) and traps including the Kernel interrupt, Timer interrupt and RTCC interrupt if applicable.
- Zone 1 contains all fibers including the scheduler.
- Zone 0 contains all priority based threads[1].

Zone 2-15	Exceptions and traps
	User interrupts
	Q▪Kernel™ Timer and RTCC interrupt
Zone 1	Priority fiber with priority 4 to 1
	Queued fibers First In First Out (FIFO)
	Q▪Kernel™ scheduler and notification fibers
Zone 0	Threads with priority from 250 to 1
	Idle thread
	Switch processor in Sleep or Idle mode

[1] Some systems use the word tasks instead of thread. We think that the word thread is more accurate. A task (amount of work) can be implemented as a thread or fiber.


Interrupts are handled based on their priority. ISR can use the interrupt stack.
User Interrupts
ISR
Kernel Timer
ISR
Kernel RTCC
ISR
Kernel Interrupt
ISR
Depending on the state of the kernel, some functions must be delayed.
See API documentation
ISR Enabled Functions
Zone 2
Kernel and Scheduler
Priority Fiber Control
Queued Fiber Control
Expired RTCC Control
Expired Timer Control
Thread Scheduling
Fiber
Fiber
Fiber
Fiber
Fiber
Fibers run priority based and use the interrupt stack
Highest priority
Lowest priority
Zone 1
Zone 0
User threads
Idle thread
User Thread
User Thread
User Thread
User Thread
Idle Thread
Higher Priority
Lowest Priority
Threads run in parallel and require their own stack
ISR provided by the developer
Fiber provided by the developer
Thread provided by the developer

3.1. Segmented Interrupt Architecture

Q-Kernel uses the Segmented Interrupt Architecture. Most competing products use the unified interrupt architecture. The difference is how the RTOS handles critical sections. A critical section is a sequence of code that must execute atomically.

The Unified Interrupt Architecture allows services to be called from interrupts but the RTOS has to disable interrupts around critical sections to avoid non-atomic changes to data.

Services in the Segmented Interrupt Architecture will always be called outside a critical section or will execute atomic. Non-atomic services that are called from an ISR while a critical section is executed are deferred with Deferred Interrupt Handlers (DIH). There is no need to disable interrupts.

Q-Kernel never disables interrupts.

This architecture provides excellent interrupt response, simplifies the development and provides Zero Interrupt Latency with full thread integration[2] and communication and prevents interrupt jitter. The architecture permits peak interrupt loads to be handled without the loss of information, which allows for tight integration between interrupts and fibers.

3.1.1 Zero Interrupt Latency RTOS[3]

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. The interrupt request signal needs to be synchronized to the CPU clock. The CPU will typically complete the current instruction. The modern hardware architectures use instructions that are single cycle or double cycle and have fixed 4 to 8 cycle interrupt latencies.

The most important delay is because the interrupts are disabled. RTOS based on the Unified Interrupt Architecture temporarily disable the interrupts to protect critical sections including thread switching.

Q-Kernel is a Zero Interrupt Latency RTOS

[2] Some competitive systems claim Zero Interrupt Latency but do not allow the interrupt handler to communicate with the threads and fibers or even ready waiting threads, which defies the purpose.

[3] Zero interrupt latency in the strict sense is not possible. What we mean when we say "Zero Interrupt Latency RTOS" is that there is no added latency to interrupts because we use an RTOS. Some MCU's don't implement the required atomic instructions and interrupts are disabled between a few instructions adding a few cycles to the latency.

Q▪Kernel™ never[4] disable interrupts, so Q▪Kernel™ will have the same worst-case interrupt latency as a system running without the Q▪Kernel™.

3.1.2 Interrupt Jitter

Jitter is the variation in interrupt latency. Disabling interrupts always leads to interrupt jitter, so the Unified Interrupt Architecture can never prevent interrupt jitter. Modern processor architectures use many registers so context switches cost more time and interrupts are disabled longer. For some applications, this can cause problems like lost interrupts.

Q▪Kernel prevents interrupt jitter

3.1.3 Dual-Mode and the Segmented Interrupt Architecture

Every Dual-Mode RTOS employs a Segmented Interrupt Architecture but not all RTOS with a Segmented Interrupt Architecture are Dual-Mode. Segmented interrupt architecture requires a Deferred Interrupt Handler (DIH), which is part of a Dual-Mode RTOS. A real Dual-Mode RTOS specifies Deferred Interrupt Handlers and implements a number of fiber mechanisms, like spawning fibers, fiber priorities and API-fiber communication.

3.2. Tick-Less Operation

Every RTOS, including Q▪Kernel™, requires mechanisms to support delay and time-out functionality. Every function that has the potential to wait can specify a timeout, which is used to specify the maximum time a thread is willing wait for a resource to become available. If the specified period expires and the resource is not available, the thread will return to the caller indicating that it timed-out. For this functionality and delay functions to work the RTOS must keep track of time.

Most RTOS use a timer interrupt to keep track of the timing of the RTOS. This interrupt is called the tick and fires every 100 µSec to one mSec.

The Q▪Kernel™ architecture provides Tick-Less operation. The system calculates the expiration time for every wait request and programs the timer accordingly. Tick-Less operation is more complex but has a number of advantages compared to non-Tick-Less operation.

- The granularity of the timer is much finer. Q▪Kernel™ uses a granularity of only one µSecond.
- There is no processing of ticks so the power consumption will be smaller. Q▪Kernel™ combines Tick-Less operation and other low-power functionality to produce an RTOS that consumes minimal power.

[4] Some MCU's don't implement the required atomic instructions and interrupts are disabled between a few instructions adding a few cycles to the latency.

- A non-Tick-Less RTOS uses the tick for all its timing operations and this will limits the maximum wait time. The maximum wait time with a 100 µSec tick is around 2 days. Q▪Kernel™ supports maximum wait times that are greater than 60 years.

3.3. Two timing sources

The Q▪Kernel™ architecture provides two sources for timing. Short times are based on the processor clock and or timer and long times are based on a Real Time Clock and Calendar (RTCC). Q▪Kernel™ uses a separate clock source for long times because the processor clock is very inaccurate over long times and there are more opportunities to save power. The short times are implemented by a 32-bit timer, called the kernel timer and the long times are based on the RTCC.

Short times are based on one cycle up to 2 billion cycles. A frequency of 100MHz provides a maximum wait time of 20 seconds. The long wait time granularity is one second and the maximum wait time is more than 60 years. There is an overlap in the one to 20 second range. We advise to use the short time up to the 10 seconds range. In this range the timer is more accurate because the granularity of 1 Sec is large compared to the wait time.

The RTCC is default implemented as an emulated RTCC. The developer can implement its own RTCC that is more effective or provides a better power savings. Working examples, based on timers and a hardware RTCC are included. If included in the application that code will be executed for optimal performance or power savings. See the examples in yRtcc2.c and qRtcc3.c.

Q▪Kernel will emulate the RTCC if there is not one available

3.4. Resource Usage

Q▪Kernel™ uses a number of resources to operate. These resources cannot be disabled or manipulated directly by addressing registers or allocating memory. Only one resource is required, the kernel interrupt. All other resources are optional. Resources are specified in the configuration.

3.4.1 Kernel Interrupt (required)

Q▪Kernel™ is activated by the kernel interrupt. This interrupt is a priority 1 interrupt and will be used exclusively by Q▪Kernel™. The interrupt can be configured and is port specific. Q▪Kernel™ requires nested interrupt priorities, meaning that interrupts can be interrupted by higher priority interrupt and interrupts should not be disabled[5].

3.4.2 Kernel Timer (required)

Q▪Kernel™ requires[6] a 32-bit timer for short time functions. This interrupt is a priority 2 interrupt and will be used exclusively by Q▪Kernel™. The interrupt is configurable and port specific. Priority 2 is the highest priority used by Q▪Kernel™ which means that zero interrupt latency and jitter free operation can be accomplished with interrupt levels 3 to 7.

3.4.3 Kernel RTCC (optional can be emulated)

The kernel RTCC can be implemented in hardware, software or emulated and is optional.

- Emulated RTCC This type of RTCC requires the kernel timer.
- Timer1 RTCC If a Timer1 RTCC is specified; the timer will be used to provide the 0.5 second interrupt. The device will be used exclusively by Q▪Kernel™ and will use interrupt priority 2.
- Hardware RTCC If a hardware RTCC is specified the RTCC will be used exclusively by Q▪Kernel™ and will use interrupt priority 2. Q▪Kernel™ will control the hardware RTCC with the exception of the calibration.
- External RTCC If an external RTCC will be used the external device has to communicate with the processor. Please contact QuasarSoft Ltd. to discuss the options.

If long wait times and real time clock information is not required this resource does not have to be configured.

[5] Q▪Kernel™ never disables interrupts but the developer can disable interrupts for short times to do a specific task like programming the hardware or programming flash. We do not recommend it though and if it is required keep the time as short as possible. Interrupts should not be disabled after the start of Q▪Kernel qKrnStart().

[6] Q▪Kernel™ can work with a 16-bit timer. Please contact QuasarSoft for more information.

4. Interrupt Service Routines (ISR)

In real-time systems, Interrupt Service Routines should be kept as short as possible. Non-essential interrupt service code should be deferred for optimal performance. Interrupt handlers should be divided into two parts: the First-Level Interrupt Handler (FLIH) and the Second-Level Interrupt Handlers (SLIH). FLIHs are also known as hard interrupt handlers or fast interrupt handlers, and SLIHs are also known as slow/soft interrupt handlers. Another name for this mechanism is "Top and Bottom halves".

The job of a FLIH is to quickly service the interrupt and to record critical information, which is only available at the time of the interrupt, and schedule the execution of a SLIH for further long-lived interrupt handling. The FLIH (ISR) reads or writes date to hardware, and then it schedules the handling of new information at a later time to the SLIH (fiber or thread) that communicates the information to the rest of the system. See this wiki article.

Keeping the interrupts as short as possible is important because the underlying hardware blocks interrupts with the same or lower priority until the ISR returns. The highest interrupt level blocks all other interrupt activity.

While dividing Interrupt Service Routines in two parts provides the best performance, Q▪Kernel™ also support the more traditional approach where all work is done in the ISR.

Q▪Kernel simplifies writing ISR's

Q▪Kernel™ does not impose any restrictions to ISRs. There are no required prologue or epilogue, like EnterISR() and ExitISR(). The developer can use local variables in the ISR, call other functions that are unavailable at compile time and use almost all[7] Q▪Kernel™ functions.

4.1. Keep ISR short

The essence of a good performing application is to keep interrupts as short as possible. This is so important because one interrupt can block other interrupts. Interrupts at the highest level block all other activity. By keeping all interrupts short the chance of missing one because very small.

The standard way to keep interrupts short is to interact with the hardware in the interrupt and do the rest of the work in a thread or fiber. Q▪Kernel™ has several ways to do this. All signal functions can be called from an interrupt and for very high loads fibers can be spawned. This is very useful because in that case the priority is not related to a thread.

[7] All functions with the exception of functions that want to wait.

4.2. Interrupt Stack

Q-Kernel™ reduces the total stack requirements significantly with the use of an interrupt stack. Without the interrupt stack, stack requirements of the ISR must be added to each thread stack because it is unknown, which thread is active when the interrupt occurs.

There are two ways to implement the interrupt stack.

- The Post-Push implementation switches to the interrupt stack after saving the registers. The registers are saved on the thread stack and the local variables and stack space for functions are stored on the interrupt stack.
- The Pre-Push implementation stores both the registers and the local variables on the interrupt stack.

Q-Kernel™ implements the pre-push stack implementation. The following example describes common interrupt requirements for the PIC32. Saving registers for a function call requires approximately 100 bytes, and the space for local variables and other function calls is approximately 80 bytes.

For an application with 4 interrupt levels and 15 threads the stack requirements are:

- No Interrupt stack (100+80) * 4 = 720 bytes per thread, and 10800 bytes for 15 threads.
- Post-Push 100*4 = 400 bytes per thread, and 6000 bytes for 15 threads plus 4*80 = 320 bytes for the interrupt stack. Total = 6320 bytes
- Pre-Push (100+80) * 4 = 720 bytes for the Interrupt stack. There are no thread stack requirements so the total = 720 bytes.

So the Q-Kernel™ implementation save more than 10kb compared to competitor's without an interrupt stack and almost 6Kb compared to competitors with a post-push interrupt stack.

Q-Kernel "Pre-Push Interrupt" Stack reduces RAM requirements significantly

Fibers, the scheduler and all ISRs plus all the functions that are called from those components use the interrupt stack.

The size of the interrupt stack is influenced by the following:

- Compiling with the code optimizer will significantly limit the stack requirements. If a compiler is used with a limited code optimizer, select the option that disables the frame pointer.
- Nesting interrupts add to the required interrupt stack size.
- Fibers use the interrupt stack. Because all activity is controlled and synchronized in the scheduler, only the function with the largest stack requirements need to be considered.

The best way to determine the size of the interrupt is to start big, like 500 bytes. The function qKrnStack() returns the stack space in use in bytes.

While the interrupt stack minimizes the memory consumption there is overhead involved in stack-switching. For that reason the interrupt stack is optional.

In most cases it is better to use the interrupt stack. The only exception is when the extra delay introduced by the stack switching is not acceptable.

4.3. Q-Kernel™ ISR

Writing an ISR which uses the interrupt stack is very simple. Write the ISR as if it is a normal function. The stack switching is done automatically, just concentrate on the functionality.

The following example specifies how to set up a TMR2 interrupt. The function qISR() will create code on the fly to define the interrupt and execute the functionality as described below.

A PIC24 example

```
qISR(_T2Interrupt) {        // 16-bit PIC example
    _T2IF = 0;              // Clear the interrupt flag
    ……                      // Do work
}
```

A PIC32 example

```
qISR(_TIMER_2_VECTOR) {     // 32-bit PIC example
    IFS0bits.T2IF = 0;      // Clear the interrupt flag
    ……                      // Do work
}
```

The function qISR_FAST() uses the processors shadow registers and is faster if they are available. The port description specifies how much space is used and if there are any limitations.

The symbols used in the function qISR() and qISR_FAST are dependent on the port. If the compiler does not recognize them, an error will be generated. See the example below.

```
qISR(_INT1Interrupt) {
    _INT1IF = 0;          // Clear the interrupt flag
    ……                    // Do work
}

Opn.c:114:1: error: pasting "(" and "_INT1Interrupt"
...
Opn.c:114: warning: return type defaults to 'int'
Opn.c: In function 'qISR':
Opn.c:115: warning: control reaches end non-void
function
```

4.4. Native Compiler Interrupt Syntax

Sometimes the activity in the ISR is limited that it is just not worth the overhead of the interrupt stack. That is definitely the case if the ISR is simple and/or it uses a fiber that does most of the work.

Calling a function in an ISR generates so much overhead that is almost always better to use the Q·Kernel™ ISR. The compiler has no knowledge of which registers are used by functions so it has to save a lot of registers. Some Q·Kernel™ functions are implemented as macros to prevent this problem from occurring. The following example is taken from the 16-bit PIC implementation and specifies a TMR5 interrupt service routine. qFbrSpawn1() is a macro so the compiler can optimize this and will not save W0-W7 on the stack. The statistics are not adjusted so cycles are added to the interrupted thread. This is not a problem because the amount of cycles is very small.

Example[8] of an interrupt service routine

```
void __attribute__ ((__interrupt__, AUTO_PSV))
                        _T5Interrupt(void){
    _T5IF = 0;             // Clear the interrupt flag
    PORTAbits.RA5 = 1;     // Indicate to the hardware
    qFbrSpawn1();          // Spawn the priority fiber
}
```

[8] This example is based on the PIC24 and dsPIC port. The Q·Kernel implementation of other ports are similar in speed and stack usage.

The generated code looks as follows:

```
180:    void __attribute__((__interrupt__, AUTO_PSV))
                        _T5Interrupt(void) {
181:                 _T5IF = 0 ;  // Clear interrupt
flag
 02E84  A98087     bclr.b 0x0087,#4
182:                 PORTAbits.RA5 = 1;
 02E86  A8A2C2     bset.b 0x02c2,#5
183:                 qFbrSpawn1();
 02E8C  884020     bset.b 0x0838,#2
 02E8E  804001     bset.b IEC1, #2
4:                }
02E96  064000       retfie
185:
```

It takes a total of 7 cycles to execute the interrupt service routine and return to the code that was interrupted. That is less than 0.5 µSec on a PIC-24F and 175 nSec on a PIC-24H or dsPIC-33. It only uses 4 bytes on the stack for the return address. Because no registers are used there is no difference between the fast interrupt (shadow attribute) and the standard interrupt so the fast interrupt can be used for something else.

It takes only 175 nanoseconds to handle an interrupt and spawn a priority fiber

The qFbrEnqueueX() functions are also defined as macros and use a limited amount of registers. By using those functions in combination with the shadow attribute, no stack-space is used and only one push.s and pop.s are generated. This combination creates very short interrupt response times and does not use thread stack space.

Priority fibers are unique to Q·Kernel™. They are super-fast and very simple.

4.5. Q-Kernel™ Services available from ISR's

The Segmented Interrupt Architecture only allows interrupts to call services when there is no ongoing critical section or the function must be atomic. If the function is not atomic, it must be deferred when the system executes a critical section. Q-Kernel™ provides a large number of signal functions that automatically defer if required and a set of functions that operate atomic. There is no difference between atomic and deferred functions for the developer. The only difference is how they operate internally.

Some competing products us different functions per environment, like SignalISR() or SignalThread(). This creates problems if there are parts of programs that are called from ISR's and threads. For this reason, all Q-Kernel™ signal functions work in every zone, in ISR's, threads and fibers.

4.5.1 Atomic Functions

Some functions operate atomically and they do not rely on the state of the kernel (critical sections). This makes atomically operated functions very fast. Most competitive systems do not use atomic operations.

Atomic operations are available for the following services:

- Queuing function to run as fibers and spawning priority fibers. These functions are essential because they are used to start deferred functions.
- Allocating and freeing Fixed Memory Blocks.
- Allocating and freeing messages based on Fixed Memory Blocks.
- Acquiring of a semaphore without blocking.

4.5.2 Deferred Functions

Deferred functions check the state of the kernel and will execute directly or will defer the function and execute it in a fiber.

All signaling functions can be called from an ISR's and the kernel will decide if it needs to defer.

Functions that wait are not allowed to be called from ISR's.

4.5.3 List of Atomic and Deferred Functions

Function	Description	Type
qEvtSignal()	Signal an event to an event-set	Deferred
qFixAlloc()	Allocate fixed memory block	Atomic
qFixAllocClr()	Allocate fixed memory block and clear	Atomic
qFixFree()	Free a fixed memory block	Atomic
qFbrEnqueue0()	Queue a fiber with no parameters	Atomic
qFbrEnqueue1()	Queue a fiber with 1 parameter	Atomic
qFbrEnqueue2()	Queue a fiber with 2 parameters	Atomic
qFbrSpawnX()	Spawn priority fiber. (X = 1, 2, 3 or 4)	Atomic
qMsgFixAlloc()	Allocate message from fixed memory	Atomic
qMsgFree()	Free a message	Deferred
qMsgPublish()	Publish a message to all subscribers	Deferred
qMsgRead()	Read a message from a pipe	Deferred
qMsgWrite()	Write a message to a pipe	Deferred
qPipGet()	Read from a pipe without synchronization	Atomic
qPipPut()	Write to a pipe without synchronization	Atomic
qPipRead()	Read from a pipe and notify writer	Deferred
qPipWrite()	Write to a pipe and notify reader	Deferred
qSemAcquireNB()	Acquire a semaphore without blocking	Atomic
qSemRelease()	Release a semaphore	Deferred
qThrEvtSignal()	Signal an event to a thread	Deferred
qThrResume()	Resume a suspended thread	Deferred
qTmrStart()	Start a timer	Deferred
qTmrStop()	Stop a timer	Deferred

5. Fibers

A fiber is program code whose primary use is to provide a low latency means of executing in response to external interrupts. Each fiber executes at a given software priority level. The Q▪Kernel™ fiber scheduler calls each fiber when it determines that the fiber has to run. A fiber has no context[9] upon entry and must perform any required data initialization upon entry. When its operations are complete, the fiber returns to the Q▪Kernel™ scheduler without context. This is the run-to-completion execution model.

Fibers improve efficiency of high data-load or Digital Signal Processing and provide fast interrupt handling and low latency. They are optimized for cooperative scheduling and fast interrupt response, to support the tight time window once data is collected. Fibers are stateless and support a single stack to sustain the required quick context switches.

Fibers are the best mechanism for handling high data-load and interrupt load.

Some of the common properties of fibers are:

- Fibers can be started from threads and interrupt service routines.
- Fibers run in the kernel, they have a priority between the highest priority thread and the lowest interrupt handler.
- Fibers do not need a separate stack. They use the interrupt stack.
- Fibers are fast, very fast
- Fibers run always inside a critical section, so they can access shared data[10].
- Fibers can use almost all services with the exception of blocking functions.
- Fibers cannot preempt, so once they are started they finish and can only be interrupted by an ISR.

There are priority based fibers, queued based fibers and Q▪Kernel™ invoked fibers. Fibers give Q▪Kernel™ a significant advantage over the competition because interrupt and data handling is much faster.

Fibers combine low latency with excellent performance.

[9] Queued fibers do not have a context but include one or two parameters specified during queuing.

[10] Shared data can be used between threads and fibers. Data cannot be shared with interrupt service routines.

There are three types of fibers, priority fibers, queued fibers and kernel invoked fibers. They are discussed in the following chapters.

5.1. Priority Fibers

Priority fibers are the fastest mechanism in Q▪Kernel™ because they can be started from interrupts with minimum overhead. Priority fibers are functions without parameters that are created with the function qFbrCreate() and are activated with the function qFbrSpawnX () where 1 is the priority from 1 to 4.

Priority fibers are the fastest mechanism to spawn a fiber from an ISR

Priority fibers are not queued and their standard use is to call them from interrupts or other heavy data-load applications. The priority is a number from one to four. Four is the highest priority and one the lowest. Two priority fibers can't have the same priority so in total four priority fibers are supported.

See the following example:

```
// This example relies on the fact that the function
// ReadyEvent is defined as priority fiber 2 with the
// function qFbrCreatePrio before it can be used

void __attribute__((__interrupt__))INT0Interrupt(void)
{
  __INT0IF = 0;              // Clear the interrupt
  if (PORTCbits.RC3 && !PORTAbits.RA3) {
                             // Direction the same
    PORTCbits.RC5 = 1;       // Indicate to hardware
    qFbrSpawn2();            // Call the fiber
  }
  else
    PORTCbits.RC4 = !PORTCbits.RC4;// Counter hardware
}


// This is the fiber function
void ReadyEvent(void) {
    ...    //Do work
}
```

The first function is the ISR and the second the fiber. The ISR compares 2 input ports. If both conditions are true it sets another port and spawns the fiber. The qFbrSpawnX() is a macro so the interrupt function does not have to save W0 to

W7 on the stack. After the ISR is done the fiber is executed. The fiber runs before all threads and can execute most services.

5.2. Queued Fibers

Queued fibers are functions that can be queued from an interrupt. Queued fibers will be executed after all interrupts and priority fibers are processed but before the highest priority thread. Fibers can be functions with zero, one or two parameters.

The functions qFbrEnqueue0(), qFbrEnqueue1() and qFbrEnqueue2() do all the work. The size of the fiber queue must be defined in the initialization section of the system. Every element of the fiber queue consists of three pointers and is allocated from the heap during initialization. Queued fibers don't have to be created they just have to be queued. The queue has limited space. If an application generates more entries than can be processed, the queue will fill up.

Queued fibers can queue a function and the required data in one atomic operation.

Q·Kernel™ uses queued fibers in signaling functions.

```
qISR_FAST(_INT0Interrupt) {
    _INT0IF = 0;                //Clear interrupt
    LATDbits.LATD8 = 0;         //set indicator
    qFbrEnqueue1(doData,(void*)PORTE);
}                               //proces in fiber

void doData(void* dataE) {      //function executed as …
   int data = (int)dataE;       //… a fiber
   … …                          //Do work
   if (workDone)                //if done
        qThrEvtSignal(pThrDisplay,EVT_DSP);
                                //display processed data
}
```

The first function in the example is an interrupt handler that reads from port E and loads the function and data in the fiber queue. After all interrupts are handled the system processes the information in the fiber queue. It will call the fiber (doData) with the data. This function will signal the display thread. This approach guarantees that even with a very high interrupt load the sequence of the data is not lost. A high interrupt load means that the fibers are not running frequently. Because the data is queued nothing will be lost and data is still processed in sequence. The ISR uses only a few bytes of the thread stack because shadow registers are used.

Example

```
qISR(_INT0Interrupt) {
    _INT0IF = 0;                  //Clear interrupt
    qThrEvtSignal(pThrHanlde,0x0100);
}                                 //Signal the thread
```

In the example above, a thread is signaled and the queued fiber details are handled automatically and transparently by Q·Kernel™.

5.3. Q·Kernel™ Invoked Fibers

The Q·Kernel™ scheduler itself runs as a fiber so all functions called from the scheduler are also fibers. The scheduler can activate the following:

- Expired timers. These events occur when a timer object reaches its time. It will call the functions that were defined when the timer was created.
- Switch notification[11]. The function qKrnNtfSwitch() will be called every time the kernel executes a context switch. Applications can use this event to implement functionality that depends on context switching.

5.4. Stack Requirements

All fibers run after each other in priority sequence. This means that the stack requirements are based on the stack requirements of the largest fiber. When a fiber is activated by the kernel the stack has been switched from the interrupted thread to the interrupt stack. So the interrupt stack should be large enough to accommodate the kernel and the largest fiber.

5.5. Extensive Use of Fibers

Fibers are the essence of all Dual-Mode Real Time Operating Systems but caution is required. Threads are interrupted by fibers and can become non-responsive. The use of fibers should be balanced with the required thread response time. On the other hand, an application with a high interrupt load that uses threads instead of fibers will see thread starvation earlier than a fiber orientated application. It will create more thread switching and thread blocking.

The developer should balance the use of fibers with the required thread response time. In most cases, fibers can be used extensively before thread starvation.

[11] Switch notification is mostly called as a fiber but in some cases as a normal function in the pre-empted thread. This does not have many consequences because the function is called in a critical section.

The link below compares a traditional RTOS and a Dual-Mode RTOS on Blackfin hardware. It shows that the difference in performance increases if the data load increases. This article uses the name Task for a Q▪Kernel™ Thread and uses the name Thread for a Q▪Kernel™ Fiber.

http://i.cmpnet.com/embedded/europe/esejun05/esejun05p38.pdf

6. Threads

A thread consists of three parts; a data structure called a Thread Control Block (TCB) that holds information about the thread, the thread stack and the program code for that thread. The TCB is used to communicate with the thread and the stack is used to store local data for the thread.

A thread is a key component in any Q▪Kernel™ based system design. Each application consists of many threads that communicate with each other. A thread executes when the Q▪Kernel™ scheduler determines that the resources required by the thread are available and there are no threads with high priority ready to run. When it begins running, the thread has control of all of the system's resources. A thread is always in one of the following states:

- **Suspended** means that the thread is not scheduled by time but must be activated by an occurrence on an object event or be resumed by another thread. The developer can create a thread in the suspended state or ready state.
- **Wait** means that the thread is waiting for an event to occur but is also scheduled to be timed out. The Q▪Kernel™ scheduler provides short wait times, provided by the timer and long wait times provided by the RTCC. This unique approach provides accurate long waiting times and limits power consumption.
- **Ready** means that the thread is available for execution. The developer can create a thread in the ready or suspended state.
- **Running** means that the thread is scheduled by the kernel to run. Only one thread can run at a time and this is always the thread with the highest priority in the ready list.

Q▪Kernel™ supports the ability of a thread to wait for the occurrence of an event or the availability of a needed resource before continuing. Q▪Kernel™ always associates the waiting thread with a particular object relating to the blockage cause.

During normal operation, some threads will be ready to execute and some will wait for a specific event to occur. Q▪Kernel™ provides three lists to perform its operations, the ready list, the timer list, and the RTCC list. The ready list is sorted on thread priority, from high to low, and the Timer and RTCC lists are sorted on time, from small to large.

6.1.1 Multi-Threading

In this context, a thread is a function running on a microcontroller with Q▪Kernel™. Without a multi-threading RTOS, only one thread can be executed by the CPU at a time. This is called a single-thread system. A real-time operating system allows the execution of multiple threads in parallel on a single CPU. All threads execute as if they completely own the entire CPU. Multi-threading allows you to switch between sections of code, giving the appearance that the CPU is doing two or more things at once. In reality, it is doing a little bit of each in succession. It is possible to run the same code in multiple threads depending on what the code does and only if the code is reentrant. There are two types of multi-threading and Q▪Kernel™ supports both.

6.1.2 Preemptive Multi-Threading

Preemptive multi-threading is a type of multi-threading where the thread can be stopped at any time or instruction, and CPU time is switched to another thread. In other words, a thread switch occurs. In preemptive multi-threading, an interrupt from a periodic timer or from a peripheral usually causes preemptive activity. In fact, an ISR preempts execution of code while it services the interrupt. In most RTOS, it simply returns back to the point of interruption, while Q▪Kernel™ can resume execution of another thread.

6.1.3 Cooperative Multi-Threading

Cooperative multi-threading occurs when the programmer yields execution to another thread with the same priority programmatically. Cooperative multi-threading is implemented but we recommend for new project to give every thread its own priority which means that the system will use preemptive multi-threading.

6.2. Scheduling

There are different algorithms that determine which thread to execute, called schedulers. All schedulers have one thing in common – they distinguish between threads that are ready to be executed (in the READY state) and threads that are suspended (in the WAIT state) for any reason (delay, waiting for message queue, waiting for semaphore, waiting for an event, and so on). The scheduler selects one of the threads in the READY state and activates it. The thread which is currently executing is referred to as the active thread. The main difference between schedulers is how they distribute the computation time between the threads in READY state. Q▪Kernel™ implements priority-controlled scheduling.

In real-world applications, different threads require different response times. For example, in an application, that controls a motor, a keyboard, and a display, the motor requires a faster reaction time than the keyboard and display. While the display is being updated, the motor needs to be controlled. This makes preemptive multi-threading a must. Other scheduling mechanisms, like Round robin do not work for embedded systems because it cannot guarantee a specific reaction time.

In priority-controlled scheduling, every thread is assigned a priority. The order of execution depends on this priority. The rule is very simple:

The scheduler activates the thread that has the highest priority of all threads in the READY state. This means that every time a thread with higher priority than the active thread gets ready, it immediately becomes the active thread.

6.2.1 Priority Inversion

In scheduling, priority inversion is a 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.

In most cases, priority inversion can occur without causing immediate harm. The delayed execution of the high priority thread goes unnoticed, and eventually the low priority thread releases the shared resource. However, there are situations in which priority inversion can cause serious problems. If the high priority thread is

left starved of the resources, it might lead to a system malfunction or the triggering of pre-defined corrective measures, such as a watch dog timer resetting the entire system.

Q▪Kernel eliminates priority inversion with its handling of mutexes. It implements the priority inheritance algorithm to eliminate priority inversions.

Priority inversion can also reduce the perceived performance of the system. Low priority threads have less precedence because it is unimportant for them to finish promptly. Similarly, a high priority thread has higher precedence because it is subject to strict time constraints. It may be providing data to an important process, or acting subject to real-time response guarantees. Because priority inversion results in the execution of the low priority thread blocking the high priority thread, it can lead to reduced system responsiveness, or even the violation of response time guarantees.

6.3. Thread Stacks

Every thread needs its own stack and the size of this stack is defined when creating the thread. The minimum size of a thread stack depends on the application, compiler and optimization options. The stack is only used by the functions that are called from threads and interrupts that do not use the interrupt stack.

Some Real Time Operating Systems use the stack to store the thread context on the stack and some will store the thread context in the Thread Control Block (TCB). The total memory requirements are the same. Q▪Kernel™ will store the thread context in the TCB and, for that reason, will not use any space on the thread stack. All space is available for the thread.

The best way to determine the size is to start big, like 512 bytes. During the creation of the thread, the stack is initialized with all “ones”. The function qThrStack() returns the stack space in bytes. The developer can examine the stack space by comparing the stack pointer with the stack size during debugging.

The size of the thread stack is influenced by the following:

- Compiling with the code optimizer will significantly limit the stack requirements. If a compiler is used with a limited code optimizer, select the option that disables the frame pointer.
- Interrupts Service Routines that don’t switch to the interrupt stack will increase the stack space because all threads must include stack space for the interrupt. Because of the dynamic nature of interrupts you don’t know which thread will be interrupted.
- Q▪Kernel™ interrupt service routines (qISR() and qISR_FAST()) use the interrupt stack and only use a few bytes of the thread stack.

- Nesting interrupts add to the stack requirements. If there are seven levels then the worst case scenario is 40 bytes added on every thread stack. If no interrupt stack is used this number will be much higher. Thread stacks need to be increased by hundreds of bytes.

Thread stacks are automatically allocated from the memory manager. The developer only has to define the size of the stack. The stack space is returned to the pool if the thread is closed.

Some microprocessors[12] have a facility to detect stack overflow. Q▪Kernel™ will control the stack for the system and will maintain the stack overflow register so the hardware can detect stack overflow.

The library with parameter checking also checks stack overflow during every Q▪Kernel™ function call. This method can help the developer find stack overflow on processors without hardware stack detection. This method is not 100% bullet proof because stack-overflow can stop the application before a Q▪Kernel™ function has been called.

6.3.1 Shared Stack (PIC24E and dsPIC24E only)

Q▪Kernel™ can use shared stacks on processors that support EDS like the PIC24F DA family, PIC24E and dsPIC24E. The stack can be shared because the content of the shared stack is saved and/or restored into EDS during a context switch. This may look like a very inefficient process but the memory bandwidth of the PIC24 and dsPIC is so large and the optimization of the algorithm makes it very practical. The following table specifies the context switch times in cycles and time where the context switch is initiated by a wait for an object like waiting for a mutex, waiting for an event, etc. The numbers are based on a 70MIPS PIC24E and the Shared Stack is 250 bytes.

Context switch initiate by waiting on object	Cycles	Time
No shared stack at all	40	0.57 µSec
Standard Stack → Standard Stack	46	0.66 µSec
Standard Stack → Shared Stack where stack was in place	46	0.66 µSec
Standard stack → Shared Stack	313	4.47 µSec
Shared Stack → Standard Stack	46	0.66 µSec
Shared Stack → Shared Stack	313	4.47 µSec

[12] The PIC24 and dsPIC have hardware stack limiting features. The PIC32 cannot detect stack-overflow in hardware. It is the user's responsibility to allocate enough stack space.

The following table is based on a preemptive context switch where the switch is initiated by an interrupt. The numbers are based on a 70MIPS PIC24E and the Shared Stack is 250 bytes.

Context switch initiate by interrupt	Cycles	Time
No shared stack at all	40	0.57 µSec
Standard Stack → Standard Stack	66	0.94 µSec
Standard Stack → Shared Stack where stack was in place	66	0.94 µSec
Standard stack → Shared Stack	333	4.51 µSec
Shared Stack → Standard Stack	66	0.94 µSec
Shared Stack → Shared Stack	333	4.51 µSec

The tables show that switching to a thread with standard stack always takes the same amount of cycles. This has been built into the algorithm to guarantee short and fixed switching times for thread with standard stacks.

Because Q▪Kernel™ never disables interrupts response times to interrupts are not affected. The rule of thumb is to use standard stack for high priority threads and use shared stacks for low priority threads. An ideal candidate for a shared stack is the TCP/IP driver. This thread will not be often activated and the TCP/IP protocol can handle long periods of in-activity that is much longer than the context switch to a shared thread.

6.4. Thread Creation

The developer can create threads before and after the start of the kernel. It is possible to create all threads before the start of the system or just one and use that thread to start other threads on demand. Threads can be created active, in the ready state or suspended, in the wait state.

See example below:

```
pTCB MainThread;
int main() {
    qKrnInit(4000000,64,256,4); // Init the kernel
    MainThread = qThrCreate(
                 "Main",    // The thread name = “Main”
                 MainThr,   // Function pointer
                 (void*)4,  // Extra parameter
                 64,        // Stack size
                 120);      // Priority
    qKrnStart();            // Start the kernel
    return 0;               // Never returns here
}
```

Threads that are created before the system is started are not activated. After the start the thread with the highest priority will be executed. If a thread is created after the start of the system it will be created in the ready state and will be activated if it has the highest priority.

The function pointer is the function to start for this thread. The extra parameter is the input for the function. The developer must define the parameter, but it is possible to cast it to another type. The example uses an integer with the value 4.

The priority specifies the priority in scheduling. Lower values represent lower priority. The idle thread has priority 0. The priority can be between 1 and 250. First define the priorities as multiples of 10 so you can later add a thread in between the priorities without having to change the other ones.

The open function returns a pointer to a Thread Control Block (TCB). This TCB can be used to in other functions to address the thread. This is very convenient because the pointer to the TCB can be local to the thread. The open function can also be used to see if a device driver is already running.

The developer can create other threads using the same function, so effectively reusing the code. This is common practice when writing device drivers where the input is the device number. A UART driver uses the parameter to select device 1, 2, 3 or 4. If the application needs 4 UART drivers, it creates 4 threads, but it loads the code only once.

The thread itself looks like this:

```
void MainT(void *p) {// Must be void *p
  int counter;       // Define local variables
  int type;
  // First execute some initialization code
  counter = 100;
  type = (INTU)p;    // Work with the startup parameter
  … … …              // do some init work
  while (counter) { // This can also an infinite loop
    …                // Code that implements the thread
                     // Waits should be included to
                     // prevent an endless loop.
  }
  // If the threads ends here. It will close itself
}
```

If a high priority thread executes an endless loop lower threads are not scheduled. This could monopolize the processor and should be prevented. The thread should not poll but use Q-Kernel™ mechanisms to synchronize itself with hardware or other threads.

This could be one of the following functions; qEvtWait(), qMsgReceive(), qMsgSend(), qMtxLock(), qSemAcquire(), qThrSleep(), qThrEvtWait() and all open functions. These are all calls with a timeout and will bring the thread in the wait state. Threads with lower priority can run.

6.5. Thread Events

A thread contains a set of events that can be manipulated by other threads and fibers. The thread can wait on any combination of event flags in one EventSet. Thread EventSets are groups of 16 or 32 binary flags[13] 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.

Thread EventSet complement the event objects. EventSet objects can be used by many threads and fibers and multiple threads can wait on these events. Use these events only if multiple threads need to wait on the EventSet, because Thread EventSets are faster and require less overhead. Another difference is that thread EventSets are always available for a thread, where event objects have to be created.

The thread can wait on any combination of event flags in one EventSet. This is very flexible. The threads can also automatically clear the flags it's waiting for. So the event wait options are:

[13] This is dependent on the port. 16-bit systems like PIC24 and dsPIC provide 16 flags and 32-bit systems provide 32 flags in one EventSet.

- WAIT_TYPE_ALL means wait until all flags are set. This is also called the AND scenario.
- WAIT_TYPE_ALL_CLEAR means wait until all flags are set and if this situation occurs reset the flags that the thread is waiting for.
- WAIT_TYPE_ANY means wait until one of the flags is set. This is also called the OR scenario.
- WAIT_TYPE_ANY_CLEAR means wait until one of the flags is set, and if this situation occurs, reset the flag(s) that triggered this operation. So not all flags that the thread was waiting for are reset.

Signaling an EventSet is possible from an interrupt, fiber or thread. This contributes to the flexibility of the events sets.

6.6. Resuming a waiting or suspended thread

Threads in the suspended or wait mode can be activated by the qThrResume() function. The thread can be in the following modes:

- Suspend mode no object event. The thread can place itself in this mode with the qThrSuspend() function. It can be resumed by the function qThrResume() and the reason specified in the resume function will be returned to the suspend function.
- Suspend mode with object event. The thread can place itself in this mode with the wait function like qEvtWait(), qSemAcquire(), qMsgReceive(), qMsgSend(), the qXxxOpen() functions, etc. The thread cannot be resumed by the function qThrResume().
- Wait mode no object event. The thread can place itself in this mode with the qThrSleep() function. It can be resumed by the function qThrResume() and the reason specified in the resume function will be returned to the sleep function.
- Wait mode with object event. The thread can place itself in this mode with the wait function like qEvtWaitTO(), qSemAcquireTO(), qMsgReceiveTO(), qMsgSendTO(), the qXxxOpenTO() functions, etc. The thread cannot be resumed by the function qThrResume().

The qThrSleep() and qThrSuspend() function return the reason why they are resumed or -1 when they are timed out. The reason can be used as a communication mechanism between the thread that activates the thread and the waiting thread.

7. Dual-Mode RTOS

Convergent processors, like the dsPIC and PIC32, bring together MCU-type control with DSP or high dataflow functionality.

The developer that uses a traditional RTOS to run DSP or high dataflow applications will run the threads as high priority threads, higher than the control threads. Because the operations are initiated from interrupts a lot of time is used to switch between threads. A traditional RTOS will use a large part of its CPU cycles just to switch between the threads.

Q▪Kernel™ combines the traditional thread-based kernel architecture for real-time control processing with specialized fibers for DSP and 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.

Fibers will be used to accommodate the high data load or DSP functionality. The process is interrupt driven. The interrupt that drives the process should be implemented as level 2 or higher. The ISR can activate a fiber by using priority fibers, or if data exchange is required queued fibers can be used. The activation of the fiber is less time consuming then a full context switch and is significantly faster. The fiber will execute the algorithm on the data and can only be interrupted by interrupts. When all data is processed and cleaned-up, the information can be sent to threads by a pipe or a queue.

It is essential to implement the data processing in fibers because they limit the overhead of thread switching. Multiple fibers can use the DSP engine because they are processed in serial according to priority sequence. It is preferred to use multiple fibers because starting a second fiber only requires a few cycles.

An application that receives an interrupt every 100 µSec from an AD converter with a measured voltage is an example of a non DSP application. The fiber has to clean the data and then send the average of the measurements every 50 measurements to a thread. If the value is critical it has to send an alarm to another thread. Instead of waking up, the thread every 100 µSec, the fiber is doing all the work. The key here is that not every interrupt requires waking up the thread. If the fiber has to activate the thread in all cases a fiber is not very helpful.

The other alternative, doing the work in the interrupt, is not good either because the interrupt will block all other interrupts that have the same or lower priority during processing.

7.1. Using the DSP Engine

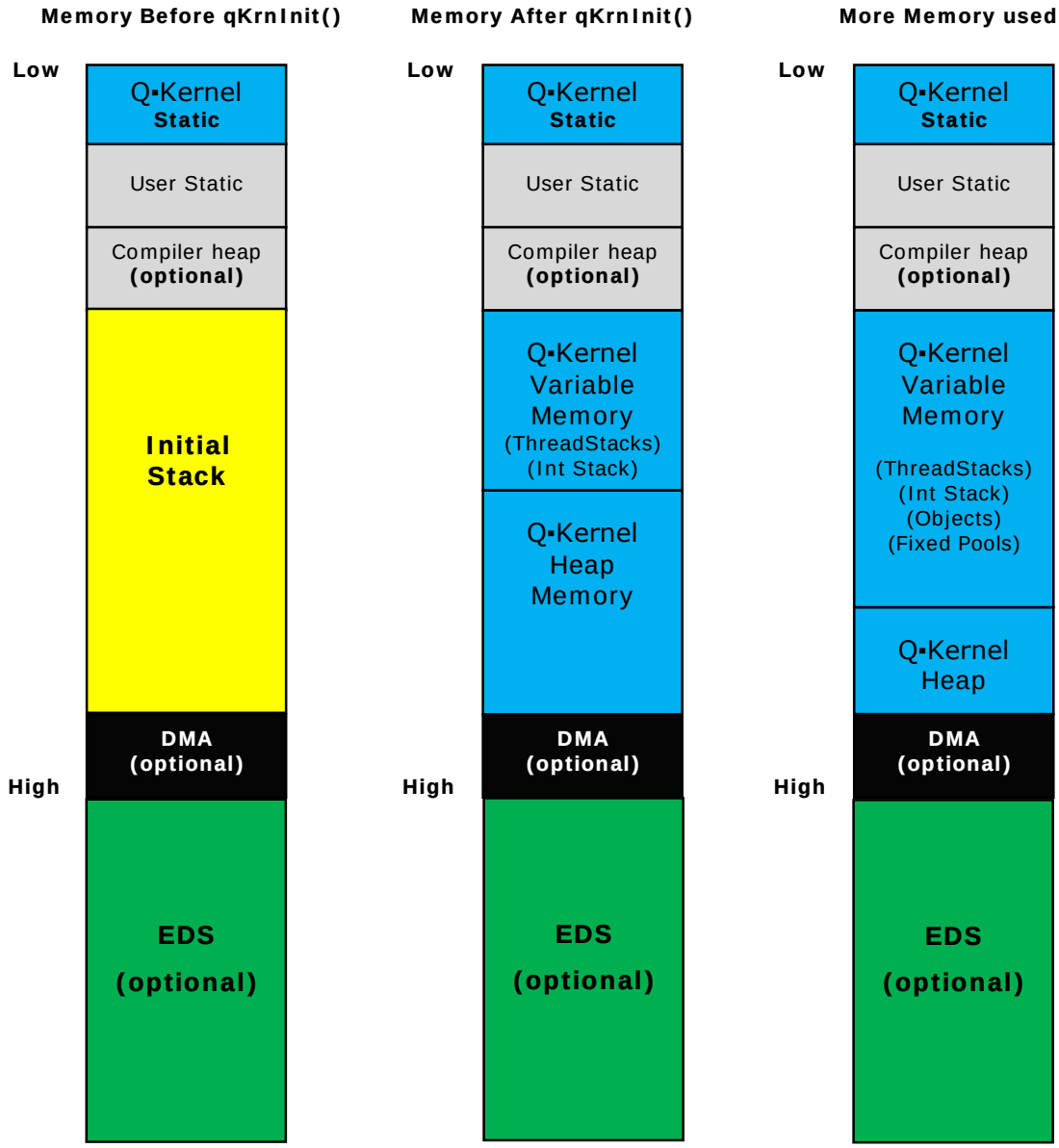
As explained above the DSP engine should only be used in fibers. Modulo and bit reversed addressing is allowed and Q▪Kernel™ supports this completely. The developer should activate the addressing mode at the beginning of the fiber and disable it at the end of the fiber.

The native compiler interrupt syntax is DSP unaware. Use the functions qDspEnter() and qDspExit() to save and restore the addressing mode. The Q▪Kernel™ interrupt syntax (qISR() and qISR_FAST()) macros are DSP aware and fully automatic.

8. Memory and Memory Allocation

When Q▪Kernel™ is initialized, it will take over the memory management. The following picture specifies the memory footprint before and after initializing Q▪Kernel™. The initial stack will be reused completely by Q▪Kernel™ to create allocate only heap memory. This memory is then converted on demand to variables and fixed memory pools. Q▪Kernel™ will implement malloc() and free() functionality through variable memory blocks.

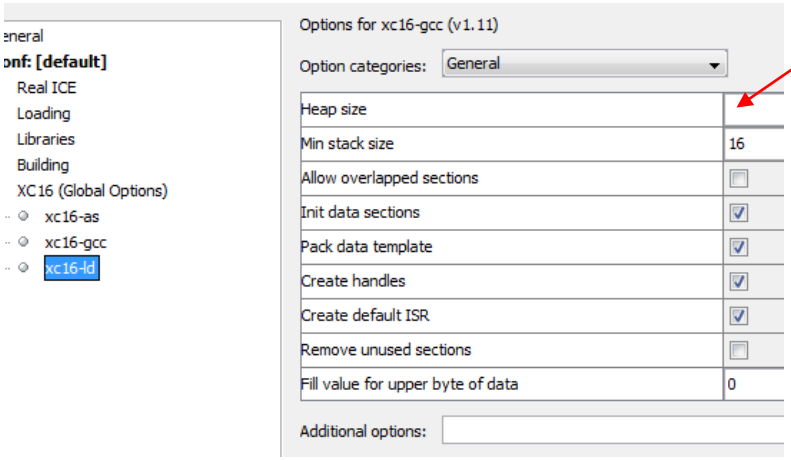
A compiler heap only need to be defined when the application wants to make use of the standard memory allocation mechanism. We do not recommend that because the standard memory allocation mechanism is not deterministic and not re-entrant.



Q·Kernel™ will only use a very small amount of static memory because it will allocate most of its memory requirements dynamically. User static variables will be allocated in User Static memory. Access to this kind of memory must be synchronized with critical sections or mutexes. The DMA and compiler heap are optional.

Q·Kernel will manage memory very efficiently.

The size of the heap can be specified in the project properties and should be zero. See the picture below.



Specify the heap size as 0 (or leave it blank) if the conventional heap is not used.

8.1. Memory Allocation

Memory allocation is even more critical in an RTOS than in other operating systems.

- Firstly, speed of allocation is important. A standard memory allocation scheme scans a linked list of indeterminate length to find a suitable free memory block; however, this is unacceptable as memory allocation has to occur in a fixed time in real-time applications.
- Secondly, memory can become fragmented as free regions become separated by regions that are in use. This is called external fragmentation and can cause a program to stall, unable to get memory, even though there is theoretically enough available. Memory allocation algorithms that slowly accumulate fragmentation may work fine for desktop machines, when rebooted every month or so, but are unacceptable for embedded systems that often run for years without rebooting.
- Thirdly, memory allocation needs to be reentrant.

Most Real Time Operating Systems provide the developer with a simple fixed-size-blocks memory allocation algorithm. This works well but is not very flexible because

the developer needs to know in advance the memory requirements, including size, and it generates a lot of work.

8.2. Memory types

Q▪Kernel™ implements four different memory mechanisms, "Allocate Only" Heap, Variable Memory Block, Fixed Memory Blocks and thread aware conventional heap. All types have their own properties.

- The “Allocate only” heap is fast but memory can’t be returned to the pool so it is inflexible.
- Fixed Memory Blocks are fast but memory has a fixed size and has to be defined before they can be used. Fixed memory blocks can be accessed by interrupts.
- Variable Memory Blocks are unique to Q▪Kernel™. They provide flexibility, are fast and reasonably deterministic.
- Conventional heap.

8.2.1 “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.

8.2.2 Variable Memory Blocks

The Variable Memory Block algorithm manages memory of any size by creating memory pools for equivalent sizes. When memory is allocated it first tries to find the memory in the correct memory pool. If there is a pool available it will simply allocate it from the head of the list. If there is no pool available it will create a pool on the fly. If there is a pool available but there is no block in the pool the block is allocated from the heap. Memory that is de-allocated (freed) will be returned to the pool it was allocated from.

Memory is allocated in multiplies of 8 bytes. So if a memory block of 44 bytes is required Q▪Kernel™ will allocate 48 bytes even if it intends to use only 44 bytes. This is called internal fragmentation. The term "internal" refers to the fact that the unusable storage is inside the allocated region.

Memory can be accessed by size or pool. Allocation by pool is significant faster because Q▪Kernel™ does not have to search for the pool. Memory allocation is 100% deterministic, so there is no difference between the allocation time of the first and last block. The access time is 12 cycles for the 16 bit implementation.

Q▪Kernel™ uses Variable Memory Blocks[14] for all its internal operations and this gives Q▪Kernel™ the ability to dispose memory that’s not used anymore. This makes the system much more dynamic. Threads and services can be created and

[14] Q▪Kernel uses the same variable memory blocks as the application with one exception. The size of the block is exact the size of the required memory and not a multiply of 8.

closed on the fly and all resources will be returned to the pool. This reduces the memory footprint.

It is better to use fixed memory blocks if the exact size and the number of blocks are known. Fixed memory blocks don't have to be a multiple of 8.

The standard heap functions malloc(), free(), calloc() and realloc() are implemented by Q-Kernel™ as variable memory but reside in the heap library. The user must include that library in the project to provide this functionality. If that library is not used, the system will use the C30-library and will use the conventional heap.

The Kernel provides a thread safe malloc() and free() out of the box

8.2.3 Fixed Memory Blocks

Q-Kernel™ provides the traditional fixed-size-blocks memory allocation algorithm. Since the blocks are fixed-size, no fragmentation will occur

Fixed Memory Blocks must be created before they can be used and they can't be changed after creation. The create function allocates the whole fixed pool with all its blocks as one big Variable Memory Block. It returns the Fixed Memory Pool address and this pool is used to allocate and free the Fixed Memory Blocks. The close function will de-allocate the whole pool and returns it to the Variable Memory Pool. It is the developers' responsibility to verify that no blocks are in use. Fixed memory blocks can be accessed from threads, fibers and interrupts because allocation and de-allocation operate completely atomic. Messages created with Fixed Memory Blocks can also be allocated in interrupts. Lack of flexibility is the main drawback of fixed-size memory pools.

Memory allocation is not 100% deterministic but very close. The allocation time for the first block is 22 cycles and the 256th block takes 38 cycles for the 16 bit implementation.

Fixed Memory Blocks are accessible from interrupts.

8.2.4 Conventional C Runtime Heap

The conventional C runtime heap or compiler heap implements dynamic memory allocation based on the compiler library function. Those functions are not thread aware, not deterministic, and can cause external fragmentation. External fragmentation is the phenomenon in which free storage becomes divided into many small pieces over time. Because they are not thread aware they can only be used by one thread. The size of the heap must be specified in the build options in MPLAB.

We advise to use the malloc() and free() functions implemented in the Q-Kernel™ heap library. They are fully thread aware and deterministic.

The conventional C heap is not reentrant and deterministic and can cause external fragmentation

8.3. Choosing Type of Memory

The following guidelines can help the developer to choose a memory type. They are in sequence of importance.

1. Always use the Q-Kernel™ heap for allocate only memory. It is fast and there is no internal or external fragmentation.
2. Use fixed memory blocks if memory is allocated and freed from ISRs or if the size and number of items is known. Fixed memory blocks don't have internal fragmentation and access is fast.
3. Use variable memory blocks only if the rules above don't apply. Variable memory blocks have internal fragmentation but are fully deterministic.
4. If there is still not enough memory available and EDS is available free-up memory by using the shared stacks for low priority threads.
5. Use the conventional heap only if variable memory does not provide a good result. The conventional heap is able to split up large memory blocks in multiple smaller blocks.

The first three rules are simple. In most cases variable memory works very well but there are cases where the conventional heap works better. One of the weaknesses of the variable memory blocks is that allocated memory of a certain size will never be re-used if there is no other request in that range. The conventional heap will re-use that, but this also means that it will not be deterministic and external fragmentation is introduced.

The best way to handle memory allocation is to start with variable memory and see if that works. This is the best situation in 99% of the cases. If this does not work, look at how much memory is allocated in pools that are not used anymore. Then see if it is possible to solve that. If that does not work, use the conventional heap. It is possible to use the conventional heap and the variable memory pool, side by side. Just keep in mind, the conventional heap may work fine in the beginning but could fail later because of external fragmentation from the conventional heap.

8.4. Using malloc(), free() and realloc()

There are two options to implement malloc(), free() and realloc(). Use variable memory or use the conventional heap. To use malloc(), free() and realloc() with the conventional heap you just have to specify the heap size as described previously. To use malloc(), free() and realloc() with variable memory blocks include the file qMalloc.c in your project. This file contains the code to use the variable memory blocks from malloc(), free() and realloc().

8.5. Allocation and De-Allocation Speed

The following table describes the operation and the speed of the operation:

Operation on memory	Speed[15]	How flexible?
Allocate from Heap	12	No free option
Allocate Fixed Memory Block	22+(N/8)	Fixed blocks must be created. They are very deterministic and depends on N[16] an interrupts.
Free Fixed Memory Block	7	
Allocate Variable Memory Blocks by pool	12 or 20[17]	100% deterministic if allocated by pool
Allocate variable memory block by size	29 to 38+2*N	Very flexible and reasonably deterministic if allocated by size. This depends on N[18] (number of pools in use)
Free variable memory block	10	

The developer has the ability to control the memory environment completely. It is possible to populate Variable Memory Pools to the required size so they can be allocated with the pool address. This gives a 12 cycle allocate speed and a 10 cycle de-allocate speed.

Under normal circumstance the Variable Memory Blocks gives the best combination of flexibility, performance and is reasonably determinist. We advise to use Variable Memory Blocks and only change this if the application makes it necessary.

If you know that the memory is never freed, use the heap. It is the fastest memory allocation algorithm.

Q-Kernel memory allocation is fast and simple.

[15] The number is the number of cycles from the start of the function including 3 cycles for the return statement. This is the exact number of cycles for the 16 bit version and a relative number for the 32 bit version.

[16] Fixed memory blocks can be used from interrupts and it is possible that an existing request is interrupted by another request. There is a possibility that the interrupted request must retry. The first block takes 22 cycles and the 32nd block takes 22 + (32/8) = 26 cycles. The Free operation is always deterministic.

[17] If there is no block in the pool the block will be allocated from the heap. The developer can create enough memory blocks during initialization to guarantee that the heap is never used. That gives a fixed 12 cycle allocation time. The Free operation is always deterministic.

[18] The N is for the number of pools in use. The mechanism first has to find the correct pool based on the size and if it does not exist it has to allocate it from the heap. So if there are 10 pools the allocation speed is 29 to 58 cycles and the free operation is a fixed 10 cycles.

8.6. Memory Functions

Function	Description
qFixAlloc()	Allocates a memory block from a fixed memory pool and returns a pointer to the memory.
qFixAllocClr()	Allocates a memory block from a fixed memory pool, clears it and returns a pointer to the memory.
qFixCreate()	Creates a Fixed Memory Pool and initializes its internal structure. Returns a pointer to the pool.
qFixClose()	Closes a Fixed Memory Pool and returns the memory to the Variable Memory Pool.
qFixFree()	De-allocate a fixed memory block and returns it to the pool.
qHeapAlloc()	Allocate memory from the heap.
qHeapSize()	Return the size of the heap.
qMemAlloc()	Allocate a memory block based on size.
qMemAllocClr()	Allocate a memory block based on size and clears the memory.
qMemAllocFast()	Allocate a memory block based on the pool.
qMemAllocFastClr()	Allocate a memory block based on the pool and clears the memory.
qMemFree()	De-allocate a memory block and returns it to its pool.
qMemPool()	Returns the pointer to a memory pool based on the size. If a pool with that size does not exist the function creates the pool.
qMemPoolAdd()	Add memory blocks to a memory pool. The memory is allocated from the heap.
qMemPoolNext()	Get from an existing memory pool the next memory pool.
qMemPoolSize()	Get the size of memory block from a memory pool.

8.7. Example Memory Allocation

The following example creates a memory pool and allocates a memory block by pool and by size:

```
typedef struct {
  int Count;
  long total;
  char insert[10];
} mydata;

void foo(){
  mydata data1,data2;// define the structure
  pMPL mpl;          // define pointer to the pool
  pMPL mpl=qMemPool(sizeof(mydata));
                     // Create pool required for …
                     // …fast allocation by pool
  data1=qMemAlloc(sizeof(mydata));
                     // allocate block by size
  data2=qMemAllocFast(mpl);
                     // allocate block by pool
  qMemFree(data1);   // free the memory
  qMemFree(data2);   // free the memory
};
```

As shown in the example, data1 is allocated by size. A pool for that size is automatically created. The function qMemPool() will return a pointer to that pool. If a pool for that size did not exist it would be created. Then another memory block (data2) is allocated with the fast method. This is completely deterministic and very fast. The last two statements free the allocated memory. It is irrelevant how the memory was created, by block or by size.

The next example shows how to create fixed memory blocks in a thread and allocates and de-allocates the block in an ISR.

```
pFIX fix;             // define pointer to the pool

void foo(){
  fix = qFixCreate(sizeof(mydata), 50);
};                    // Create pool with 50 blocks

// The ISR
qISR(_T2Interrupt) {          // 16-bit PIC example
  mydata data=qFixAlloc(fix); // Allocate memory
  ……..                      // Do work
  qFixFree(data);             // free the memory
}
```

9. Power Management

Strategies to reduce power can be particularly useful in applications that are both power-constrained (such as battery operation), yet require periods of full-power operation for timing-sensitive routines, such as serial communications, AD conversions, etc. Q▪Kernel™ delivers industry leading power management due to its Tick-Less operation and its flexible implementation.

Limiting power consumptions can be approached by limiting the clock frequency of the processor or by stopping the processor and switching to a low power mode when idle. Limiting the clock frequency is not as effective as switching to a low power mode when idle because most modern processors consume more power per cycle at a lower clock frequency. The most effective way to limit power consumption is to work at the maximum operating frequency and put the processor in sleep mode when idle. This requires an interrupt driven application without polling. Most competing products are not Tick-Less so the power savings are limited because the processor must poll[19] every 100 µSec to 1 mSec to process its tick.

Q▪Kernel is Tick-Less and provides the best power management.

Q▪Kernel™ can help[20] the developer to find when the system has processed all interrupts, fibers and threads and becomes idle and manage power.

Most MCU can operate in at least 3 power modes[21]:

- Normal mode means the processor runs at full speed and power consumption is at a maximum.
- Idle mode stops the processor completely but keeps the device clock going. While this mode is using more power than sleep mode, it can be a very good alternative because it gives the user a lot of control like disabling some of the hardware devices based on requirements.
- Sleep mode stops the processor completely and stops most of the devices. This mode consumes the least power of all power modes.

[19] Some systems disable the tick. While that limits the power consumption, it prevents timing and delays and involves the implementation of manual processes that need to be developed. Q▪Kernel provides all functionality "out of the box".

[20] Can help is an understatement. It is difficult to effectively switch to a low power without an RTOS like Q▪Kernel.

[21] The deep-sleep mode of some processors is not supported. This mode is for very small applications that have very low wake-up requirements and operate with limited amounts of RAM.

Q·Kernel™ is not only optimized to find the moment the system is completely idle but also detects if there are no future outstanding activation requests that would be influenced by the power mode. The system also guarantees that the power mode will not switch when an interrupt is ready to be processed. This could cause a race condition and would prevent the interrupt from being processed in time.

Q·Kernel prevents race conditions between interrupts and power mode switching.

The power mode is controlled by permitting or preventing Idle or Sleep modes. The functions qPwrPermitIdle(), qPwrPermitSleep(), qPwrPreventIdle() and qPwrPermitIdle() control this.

Q·Kernel power management is simple.

9.1. Interrupt Response Time in Low Power Mode

The interrupt response time provided by the processor is related to the power mode. In normal power mode the interrupt response time is a few cycles. In idle and sleep mode there is a wake-up delay that can delay the interrupt response significantly. Please refer to the processor data sheet for the specific wake-up delay times.

10. Statistic Services

Q-Kernel™ provides statistic services and thread switching can be tracked by using the switch notification function. Statistics are more useful over time.

10.1. Switch Notification

Switch notification can be enabled with the functions qKrnSwitchNotificationOn() and disabled with the off variant. If switch notification is on, the system will call qNtfSwitch(). The two arguments are the current thread and the next to run thread. The developer can write code to solve difficult to find issues. Because the notification function is a normal function, running as a fiber the developer can code extensive tracking mechanisms for debugging purposes. This type of tracking is intrusive.

Switch Notification is intrusive.

This functionality can also be used to make variables thread aware.

10.2. Statistics

Q-Kernel™ provides the developer of an application with thread and fiber statistics. The service will measure the total number of cycles used by every thread, individual and for the fibers combined. This information is used to calculate how much CPU time is used by every thread, individually and fibers in total.

The developer has to configure statistics and the gathering of statistics starts with the execution of qKrnStatOn(). The algorithm used does not use much CPU time itself and the statistics are very accurate. Inaccuracy is introduced by interrupts[22] and closing a thread[23].

Statistics are gathered in the core of the kernel during thread switches and fiber activation and adds about 100 cycles to context switches and fiber activation.

Switch Notification Function	Description
qKrnSwitchNotificationOn()	Enables switch notification
qKrnSwitchNotificationOff()	Disables switch notification
qKrnNtfSwitch()	Function provided by developer and called from the scheduler during a thread switch.

[22] Time spend in interrupts is counted to the tread or fiber that was interrupted. This is no problem because interrupts are very light and don't use much CPU-time. The design philosophy of Q-Kernel is to keep interrupts as short as possible and do the bulk of the work in fibers.

[23] If a thread is closed within a measuring period the time spend by the thread is counted as it was spend by the fibers.

Statistic Function	Description
qFbrStatCycles()	Returns the total number of cycles since the start of the statistic gathering for fibers and scheduler.
qKrnStatOff()	Ends statistic gathering.
qKrnStatOn()	Starts statistic gathering.
qThrStatCycles()	Returns the total number of cycles since the start of the statistic gathering for the specified thread.

11. Services and Objects

Services are available for communication, synchronization, memory management, threads and fibers.

Almost every service uses an object to identify the service and to communicate with the service. The objects have a structure that is partly available for the application but are mostly used internally by Q·Kernel™.

The different object types are listed below with their data type and prefix.

Services	Structure	Pointer to Structure	Functions
Critical Sections	None	None	qCrt.....()
EventSets	EVT	pEVT	qEvt.....()
Errors	ERR	pERR	qErr.....()
Fibers	None	None	qFbr.....()
Managed Messages	MSG	pMSG	qMsg.....()
Variable Memory (pools and items)	MPL	pMPL	qMem.....()
Fixed Memory (pools and items)	FPL	pFPL	qFix.....()
Mutexes	MTX	pMTX	qMtx.....()
Pipes	PIP	pPIP	qPip.....()
Queues	QUE	pQUE	qQue.....()
Semaphore	SEM	pSEM	qSem.....()
Threads	TCB	pTCB	qThr.....()
Timers	TMR	pTMR	qTmr.....()

All objects are type safe. This means that errors are found at compile time instead of run-time, which saves development time. Type safe objects do not degrade performance. Q·Kernel™ still checks the validity of the object to prevent the use of the object when it is in an invalid state.

Q·Kernel objects are type save which will shorten development time.

11.1. Dynamic Object Management and Naming

Most service objects[24] need to be created before they can be used. Q▪Kernel™ handles object dynamically so they can be returned to the pool.

The dynamic resource creation has a number of important benefits. It allows the developer to save resources, mainly RAM, because threads or other objects can be closed and resources are returned to the resource pool.

Multiple threads, fibers and interrupts use the same objects. One thread will use a queue to send messages and another thread will read that message from the same queue so they must use the same object. Traditional products store the object pointer in a global variable. This means that the developer has to synchronize the use and the creation of the object. Q▪Kernel™ solves this problem by supplying a "Create" and an "Open" function. The "Open" function waits until the object is created. We advise the use of local variables (type save pointers) to store the object (pointer).

The ability to search for an object and synchronize it with the create ones prevents global variables and simplifies development.

During creation, the object can be given an optional[25] name. Other threads or fibers can find the object by using its name. The following resource objects implement open services.

- EventSets (qEvtCreate() and qEvtOpen())
- Memory Pools[26] (qMemPool())
- Mutexes (qMtxCreate() and qMtxOpen())
- Pipes (qPipCreate() and qPipOpen())
- Queues (qQueCreate() and qQueOpen())
- Semaphore (qSemCreate() and qSemOpen())
- Threads (qThrCreate() and qThrOpen())
- Timers (qTmrCreate() and qTmrOpen())

[24] Critical Sections and fibers are exceptions because the administration is maintained within Q▪Kernel and they are always available so the developer does not have to create them and open and close functions are not required.

[25] Objects that are not named cannot be opened by another user of that object. Shared objects should always be named. Q▪Kernel open functions use the name to find the object. The developer can use the constant qNO_NAME for readability reasons.

[26] Memory pools are not identified by their name but by the size of the blocks in the pool.

The following functions are available:

- The qXxxCreate() function allocates the resource and returns a type save pointer to the caller. Before the function returns, it will ready threads that are waiting on the creation of the resource, so it is possible that the thread will be pre-empted.
- The qXxxOpen() function finds existing resources and returns a pointer to the object or waits for another thread to create the object. This way the pointers to those blocks can be local to the thread or fiber. The open requires the name of the resource to find the resource. The name can be defined as a constant string or a literal.
- The qXxxOpenNB() function tries to find an existing resource and returns a pointer to the object if it exists. If the resource does not exist, it will not block but will return a null pointer.
- The qXxxOpenTO() functions as qXxxOpen() but will limit the wait time. If the resource is not available within the wait time, it will return a null pointer.
- The qXxxClose() function returns the resources back to resource pool. The object cannot be used anymore and will be invalidated. This guarantees that the object cannot be used by accident.

The following example specifies how to synchronize the creation of a semaphore between two threads. The main thread creates the queue

```
void myThr(void *p) {     // Thread
  pQUE pQue;              // Local variable
  pQue=qQueOpen("myQue1");// Get a pointer and wait
                          // until the queue is created
  … …                     // Do work with it
}

void mainThr(void *p) {   // Thread to create queue
  pQUE myQue1;            // Local variable
  pQUE myQue2;            // Local variable
  myQue1 = qQueCreate("myQue1",20);
                          // Create a queue
  myQue2 = qQueCreate("myQue2",10);
                          // Create a second queue
  … …                     // Do work
}
```

11.2. Time-Out and Blocking Functions

All functions that communicate with services that have the ability to wait until a resource is available come in 3 variants:

- The Standard functions, like qSemAcquire(), will wait until the resource is available. It does not matter how long that will take. The thread can only be activated by another thread or fiber. This type can only be used by threads, because it requires state. Because this type does not time-out it does not use the wait-timer or RTCC which can minimize power use.
- The Non-Block functions, like qSemAcquireNB(), will always return immediately with two possible outcomes. The resource is acquired and the function returns success or the resource is not available and the function does not return success. This type can be used by threads or fibers.
- The Time-Out functions, like qSemAcquireTO(), will return if the resource becomes available or times out, if the resource is not available within the timeout time. This type can only be used by threads, because it requires state. This type can use the wait-timer or RTCC depending on the time specified. A negative value will use the real-time clock. There are some examples below.

```
qSemAcquireTO(pSem, -10);
                // Acquire the semaphore and timeout
                // after 10 Sec Use RTCC
qSemAcquireTO(pSem, 10000000);
                // Acquire the semaphore and timeout
                // after 10 seconds Use wait Timer
qSemAcquireTO(pSem, 50);
                // Acquire the semaphore and timeout
                // after 50 microseconds
```

The first two examples specify a timeout of 10 seconds. The first one uses the real-time clock and the second and third example specifies the delay in µSec.

11.3. Error Handling

Q▪Kernel™ provides centralized error handling. All errors are defined as fatal errors, in which the application can't continue. When a function returns the developer knows that the operation ended successfully[27]. Centralized error handling minimizes code complexity. These kinds of errors are often coding errors or the system reaches limitations and can't continue. It is not necessary to test the result of a function.

[27] The term successful means that Q▪Kernel did not find any condition that it cannot continue. A function which times out is in most cases not successful but Q▪Kernel defines the outcome as a success because a timeout is a valid outcome of a wait.

All Q▪Kernel™ errors have a unique code and this code identifies exactly what's wrong. All errors are documented in the reference guide. Some of the optimized builds limit error checking for performance and size reasons.

Traps like address errors, stack overflow etc. are also detected by Q▪Kernel™. Those errors are port specific but are signaled by the same notification function.

11.3.1 Application Errors

The Q▪Kernel™ error handling mechanism can be used by the developer to throw application errors. Errors from 0x0001 to 0x0FFF are reserved for application errors and can be handled the same way as Q▪Kernel™ errors. The developer has to call qNtfError(MY_ERROR) to throw the error and use the Q▪Kernel™ mechanisms.

11.3.2 Notification of Errors

Every time an error occurs, the function qErrNotify() is called. The developer must provide the qErrNotify () function. The most common and minimal function is listed below but the developer can easily extend the functionality. Caution is required because the system can be in an unknown state. The function checks if there is an error and if that is the case it will call qKrnError() to reset the system. The error can be logged after the reset and qKrnInit()[28].

```
INTU qErrNotify(INTU err) { // The function
    if (err==0) return err; // if no error return it
    qKrnError(err);         // call the kernel error …
    return err;             // … so it is logged in …
};                          // … persistent RAM
```

A simple breakpoint can be set on the entry of the notification function for debugging purposes.

The developer can also use this mechanism for its own error handling. This creates complete centralized error handling and only one error logging mechanism needs to be implemented. Also

11.3.3 Logging of Errors

Most applications require the logging of errors in some kind of non-volatile memory like flash. This would make it easier to find problems because more information is available. The problem is that the system is not in a stable state when the error is notified and it might be impossible to log the error. The solution for this problem is to call qKrnError() which logs errors in persistent[29] memory and then resets the processor. The error information is available after the initialization phase of Q▪Kernel™.

[28] See for an example the Blinky.c

[29] The term persistent means that the memory is not overwritten by the initialization code after a reset.

The error information is available in the structure qsERR and contains the following information:

- ErrorNbr This number specifies the type of error. The number has the following ranges:
  - Errors below 0x1000 are application errors and the developer is responsible for those errors.
  - Errors between 0x1000 and 0x1FFF are documented in the reference manual.
  - Errors between 0x2000 and 0x2FFF are port specific errors and are documented in this guide at the port information.
- StackType This value specifies if the interrupt stack is in use (1) or if a thread stack is in use (0).
- TrapAddress This value specifies at which address (PC) the trap occurred. This is always a 32-bit address.
- Zone This number indicates in which zone the error occurred.
  - Zero means that the error occurred in a thread.
  - One means that the error occurred in a fiber or in the Q▪Kernel™ scheduler.
  - Two and higher means that the error occurred in an interrupt handler and the number specifies the interrupt level.
- pThread This pointer specifies the current thread. If the zone is zero then this thread initiated the error. If the zone is not zero this is the interrupted thread.
- UpTime This value specifies the number of seconds the system was up before the error occurred.

Other information is port specific and is described in the port specific chapters.

Q▪Kernel™ will make a copy of the last error information during the initialization qKrnInit() and returns that. This will make it possible to log the information after the error occurred when the system is stable.

Q-Kernel provides the most complete and the best error handling in the business

11.3.4 Error Handling Example

The following code snippet saves the pointer to the error information and creates a special logging thread to log the error. If the error is logged the thread will remove itself from the system and returns the resources to the resource pool.

```
int main(void) {            // main entry point
  pERR err = qKrnInit(……); // Initialize Q-Kernel
  if (err)                  // If err start logging
    qThrCreate(0,logThr,err,256,10);
  qThrCreate(0,mainThr,0,256,5);
  qKrnStart();              // Start Q-Kernel
}


void logThr(void *p) {
  pERR err = (pERR)p;     // save the pointer to error
  … … …                   // Log the error
}                         // Exit will close the thread
```

11.4. Structures, Unions and Data Types

One of the goals of Q▪Kernel™ is to let the compiler do as much checking as possible. Structures will accomplish this and make it difficult to mix data types. The compiler will flag an error.

11.4.1 Common Structures, Unions and Defines

There are a number of structure definitions[30] in qKernel.h file that can be used to simplify programming and improve compatibility between the different versions of Q▪Kernel™.

Structure Union	Explanation
VER	Version information
DATETIME	Union specifying a bcd date-time.
INT16U	Union to separate bytes in a 16 bit unsigned integer
INT32U	Union to separate bytes and 16 bit unsigned integers in a 32 bit unsigned integer
INT64U	Union to separate bytes, 16 bit unsigned integers and 32 bit unsigned integers in a 64 bit integer

See the version structure below:

```
typedef struct sVER {  // Example 3.0-1389
  uint8_t major;       // Major build number 3
  uint8_t minor;       // Minor build number 0
  uint16_t build;      // Build number ios 1389
} VER;
typedef const VER* pVER;
```

[30] The Q▪Kernel objects are also structure definitions but are discussed later in this document.

DateTime example:

```
typedef struct sDATETIME {
  uint8_t year;     // BCD codification 0x00->0x99
  uint8_t rsvd;     // Reserved for future use
  uint8_t mday;     // BCD codification 0x01->0x31
  uint8_t mon;      // BCD codification 0x01->0x12
  uint8_t hour;     // BCD codification 0x00->0x23
  uint8_t wday;     // BCD codification 0x00->0x06
  uint8_t min;      // BCD codification 0x00->0x59
  uint8_t sec;      // BCD codification 0x00->0x59
} DATETIME;
typedef DATETIME* pDATETIME;
```

UINT64 example. The UINT32 and UINT16 are similair

```
typedef union uINT64U {
  uint8_t  uint8_0;   // Byte access (LSB)
  uint8_t  uint8_1;   // Byte access
  uint8_t  uint8_2;   // Byte access
  uint8_t  uint8_3;   // Byte access
  uint8_t  uint8_4;   // Byte access
  uint8_t  uint8_5;   // Byte access
  uint8_t  uint8_6;   // Byte access
  uint8_t  uint8_7;   // Byte access (MSB)
  uint16_t uint16_0;  // 16 bits access (LSHW)
  uint16_t uint16_1;  // 16 bits access
  uint16_t uint16_2;  // 16 bits access
  uint16_t uint16_3;  // 16 bits access (MSHW)
  uint32_t uint32_0;  // 32 bits access (LSW)
  uint32_t uint32_1;  // 32 bits access (MSW)
  uint64_t uint64;    // 64 bits access
} UINT64;
typedef UINT64* pUINT64;
```

12. Critical Sections Services

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 or fiber. 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 and fiber activation. This is an effective low cost way of implementing a critical section. It takes 1 cycle to disable the thread switch and it will take a few cycles to resume thread switching, including testing to see if a thread switch request was queued. 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.

```
qCrtEnter();              // Enter the critical section
if (GlobalData-- == 4)    // Do work with global data
    GlobalData = 100;     // Do more work
qCrtExit();               // Exit the critical section
```

Another method to implement a critical section is a mutex. Mutexes are slower than critical sections but only suspend the thread that wants to access the critical resource. Critical sections block all threads switching but are much faster. For this reason critical sections should be kept as short as possible.

Critical Sections should not run longer than a few hundred µSeconds.

It is not necessary to use critical sections within fibers because fibers can never be interrupted by threads. It is as if fibers always run within a critical section.

Critical sections can be nested up to thousands of levels. The developer only has to keep track that every qCrtEnter() has a qCrtExit() in all flows.

12.1. Critical Sections and Interrupts

As stated above critical section cannot be used to share data between ISR's and other parts of the system. Most Real Time Operating Systems use critical sections that disable interrupts for synchronization. Q-Kernel™ implements the "Segmented Interrupt" architecture to prevent disabling of interrupts. The synchronization can be done in any fiber type but most commonly used is the queued fiber because it allows data exchange.

Critical Sections cannot be used to protect data that must be shared with Interrupt Service Routines

13. EventSet Services

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 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.

EventSet complements the event features of threads. Thread events belong to the thread and only that thread can wait on the thread. Multiple threads can wait on these events. Use these events only if multiple threads need to wait on the EventSet, because thread EventSets are faster and require less overhead.

A thread uses the qEvtCreate() function to create an EventSet object. The creating thread specifies the initial state of the flags and also specifies a name for the event object. Threads can open an existing EventSet object or wait for an EventSet object by specifying its name in a call to the qEvtOpen() function.

Multiple threads can wait on any combination of event flags in one events set. This is very flexible. The threads can also automatically clear the flags it's waiting for. So the event wait options are:

- WAIT_TYPE_ALL means wait until all flags are set. This is also called the AND scenario.
- WAIT_TYPE _ALL_CLEAR means wait until all flags are set and if this situation occurs reset the flags that the thread is waiting for.
- WAIT_TYPE _ANY means wait until one of the flags is set. This is also called the OR scenario.
- WAIT_TYPE _ANY_CLEAR means wait until one of the flags is set and if this situation occurs reset the flag(s) that triggered this operation. So not all flags that the thread was waiting for are reset.

When a thread signals an events set, any number of waiting threads that specify the same event object 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.

Fibers can also be used with EventSet services. They can use all functions with the exception of wait type functions.

Signaling an EventSet is possible from an interrupt, fiber or thread. This contributes to the flexibility of the events sets.

Events can be signaled from Interrupt Service Routines (ISR)

13.1. EventSet Functions

Function	Description
qEvtClear()	Clear event flags in an EventSet
qEvtClose()	Close an EventSet
qEvtCreate()	Create a new EventSet
qEvtOpen()	Return the existing EventSet or wait until an EventSet with that name is created.
qEvtOpenNB()	Return the existing EventSet with that name or return null if the EventSet with that name does not exists.
qEvtOpenTO ()	Return the existing EventSet or wait until an EventSet with that name is created with time out.
qEvtSignal()	Sets events flags in an EventSet and signals
qEvtWait()	Wait for ALL or ANY event flag to be set in the EventSet
qEvtWaitNB()	Check for ALL or ANY event flags to be set in the EventSet
qEvtWaitTO()	Wait for ALL or ANY event flag to be set in the EventSet with time-out

13.2. EventSet Example

14. Mutex Services

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. For example, to prevent two threads from writing to a serial LCD at the same time, each thread waits for ownership of a mutex object before executing the code that writes to the serial LCD. After writing, the thread releases the mutex object.

Fibers can't lock mutexes because mutexes are owned by threads.

A thread uses the qMtxCreate() function to create a mutex object. The creating thread can request immediate ownership of the mutex object and can also specify a name for the mutex object.

Threads can open an existing mutex object or wait until the object is created by specifying its name in a call to the qMtxOpen() function.

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.

If a thread owns a mutex only that thread can unlock the mutex. This means that it is impossible to unlock the thread from an ISR or fiber.

14.1. Priority Inversion

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.

Multiple related locks can be a problem in any design. Q-Kernel™ implements the priority inheritance algorithm but that does not guarantee that all priority inversion and dead-lock problems are solved.

The best strategy for solving priority inversion is to design the system so that inversion can't occur. Although priority inheritance prevents unbounded priority

inversion, the protocol does not prevent bounded priority inversion. Priority inversion, whether bounded or not, is inherently a contradiction. You don't want to have a high-priority thread wait for a low-priority thread that holds a shared resource.

14.2. Alternatives

Note that critical sections provide a similar service to that provided by mutex objects, except that a critical section prevents thread switching while a mutex blocks thread execution of the threads that are waiting for the mutex. Mutexes are more selective compared to critical sections but they are significantly slower. Use mutexes if the time to execute the critical section exceeds 100 µSec. Critical sections are the best solution to provide shared access to memory. Another interesting fact is that fibers always run as if they run inside a critical section so you don't have to use critical section.

14.3. Mutex Functions

Function	Description
qMtxClose()	Close an existing Mutex
qMtxCreate()	Create a new Mutex
qMtxLock()	Lock an existing Mutex
qMtxLockNB()	Lock a existing Mutex without blocking
qMtxLockTO()	Lock a existing Mutex with time-out
qMtxOpen()	Return the Mutex by name or wait until a Mutex with that name is created.
qMtxOpenNB()	Get an existing Mutex or wait until a Mutex with that name is created.
qMtxOpenTO()	Return the Mutex by name or wait until a Mutex with that name is created with time out.
qMtxOwner()	Returns the owner of the mutex or null if nobody owns the mutex.
qMtxUnlock()	Unlock a locked Mutex.

14.4. Mutex Example

15. Semaphore Services

Conceptually, a semaphore maintains a set of permits. When a permit is acquired the number of permits is decremented if no permit is available. A release of a permit adds a permit, potentially releasing a blocking acquirer. If more than one thread is waiting on a semaphore, the thread with the highest priority is selected.

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 that 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.

Permits can be released from an Interrupt Service Routines (ISR)

Semaphores are the fastest synchronization mechanism. Semaphore functions:

Function	Description
qSemAcquire()	Acquire a semaphore
qSemAcquireNB()	Acquire a semaphore without blocking
qSemAcquireTO()	Acquire a semaphore with timeout
qSemClose()	Close an existing semaphore
qSemCreate()	Create a new semaphore
qSemOpen()	Open an existing semaphore or wait until it is created
qSemOpenNB()	Open an existing semaphore no blocking
qSemOpenTO()	Open an existing semaphore with time-out
qSemPermits()	Returns the number of permits
qSemRelease()	Releases a permit

15.1. Semaphore Example

The following example opens a semaphore and acquires a permit on the semaphore with a 2-second timeout.

```
//-----------------------------------------------------
//  Thread code that waits for an access point
//-----------------------------------------------------
  pSEM pSem
  pSem = qSemOpen("APS");      // Open expect that
                               // semaphore is created
    … … … …                    // Do other things

  if (!qSemAcquireTO(pSem,2000000)) {
    … … … …                    // No access in 2 sec
  }                            // Handle error

//If here we have the access point so do work
    … … … …                      // Do other things
```

The following code releases a permit on the semaphore which will activate the waiting thread.

```
qISR(_INT0Interrupt) {
  _INT0IF = 0;             // Clear the interrupt
  … … …                    // Create the access point
  qSemRelease(pSemAP);     // Release an access point
}
```

16. Pipe Services

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

Fast ISRs normally operate in the 1 to 10 µSec time frame. A thread operates 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.

A good example is an UART driver. A thread or fiber writes information in the pipe and the transmit ISR reads the data from pipe and writes it in the hardware register to transmit the character. At a baud rate of 1Mbit a transmit interrupt is fired every 10 µSec.

If the buffer is (almost) empty the ISR readies a waiting thread and can then write new information in the pipe. If the size of the buffer is 100 and the ISR readies the waiting threads after 80 characters the thread will switch every 800 µSec. On a 16 MIPS processor this generates a load of only 2.5%

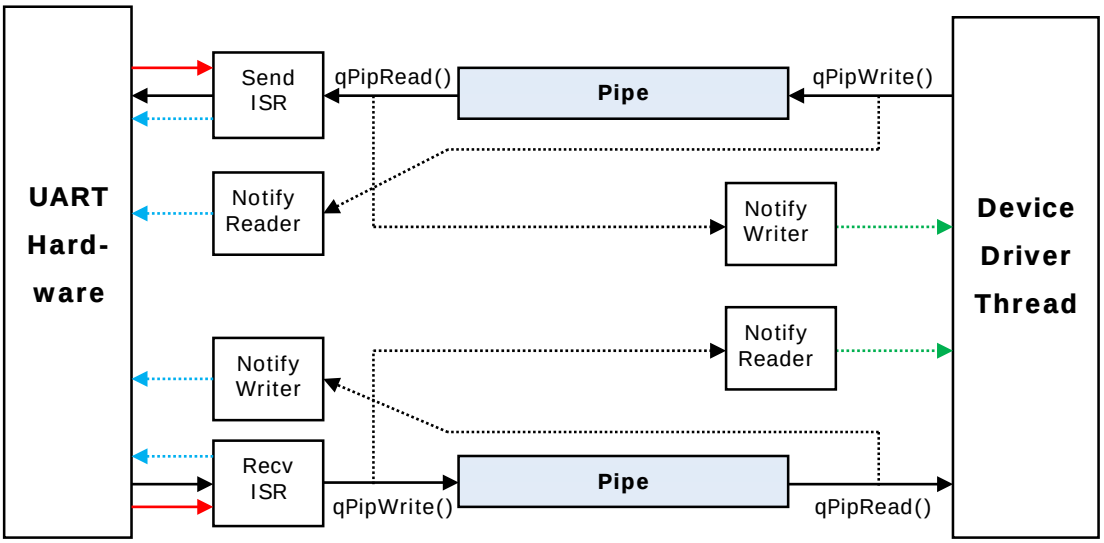
Pipes offer excellent performance for integration between ISRs and Threads

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 situations. Sometimes a reading thread must be activated as soon as one element is put in the pipe. In other cases 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.

So if a reading thread needs to be activated when the pipe is 80% full the "Notify Reader" function tests every time it is called if the 80% is reached. If the pipe is less than 80% full it does nothing otherwise it activates a waiting thread.

This type of synchronization can minimize the number of context switches and can better work in conjunction with hardware queues and DMA. Q-Kernel™ has a significant advantage over competitive products because the implementation is more flexible and minimizes context switches.

The following block diagram shows how a UART driver would be implemented with pipes and thread events:



On the left side are the UART hardware registers and on the right side is the thread that functions as a driver. There is a pipe for sending information and a pipe for receiving information. The two ISRs on the hardware side are handling interrupts (red-line). The blue lines set hardware registers and the green lines ready the thread. The solid lines specify how data moves.

The block diagram handles four scenarios that are discussed below:

- The thread writes with qPipWrite() data in the write-pipe. qPipWrite() will execute the notification reader which controls the hardware and can enable interrupts. By doing so the UART will fire the transmit interrupt. If the thread cannot write the data it waits for a writer event.
- The UART fires the Transmit interrupt and activates the send ISR. This ISR will read information with qPipRead() and that function notifies the notification writer. This function will signal the thread event so the thread will be readied if it was waiting for the event so it can keep writing.
- The UART receives a character and executes the Receive ISR. This ISR read the data from the hardware and writes it to the pipe with qPipWrite(). This function notifies the reader and signals the thread event so it will activate the thread if it was waiting for the pipe to be filled.
- The thread reads with qPipRead() function and if the pipe is empty the system will wait for the read event. This event will be signaled by the writer.

As seen in the previous example one thread can handle multiple events. It can just wait on ANY event and will become active if there is something to do. Q▪Kernel™ has a significant advantage over competitive products because one thread can not only handle sending and receiving but can handle multiple UARTS. Most other competitive products require two threads to handle the scenario above and require 8 threads to handle four UARTS. This saves RAM, Flash and limits context switching.

One Thread can handle multiple pipes, limiting RAM, Flash and context switches.

16.1. Multiple Readers and Writers

Pipes support multiple readers and writers like threads, fibers and ISRs. This make them unique compared to the competition. The following scenarios are supported:

- Multiple Threads can read the same pipe
- Multiple Threads can write the same pipe
- Multiple Fibers can read the same pipe
- Multiple Fibers can write the same pipe
- Multiple ISR's with the same priority can read the same pipe
- Multiple ISR's with the same priority can write the same pipe

All other scenarios are not supported like a Thread and ISR both reading, Thread and Fiber are both reading, Fiber and ISR are both reading or ISR's with different priorities are reading. The same scenarios for writing are also not supported.

The priority and type of reader and the priority and type of writer of the same pipe can be completely different. Examples of these cases are:

- A Thread can be the reader and an ISR's the writer or vice versa
- An ISR can be the reader and another ISR with a different priority can be the writer or vice versa.
- An Fiber can be the writer and a Thread can be the reader or vice versa
- An Fiber can be the writer and an ISR can be the reader or vice versa

Multiple readers and writers introduce some challenges for the RTOS because it needs to control the pipe atomically. Q▪Kernel™ solved this by creating a block oriented approach that will work for bytes, or integers and complex structures. The whole block will be transferred in one atomic operation and guarantees that structures are transferred as a whole. Competing products work only with arrays of bytes or integers and can't guarantee that the sequence of bytes and integers is exactly the same with multiple writers.

While Q▪Kernel™ guarantees that a block is transferred in one atomic operation it does not guarantee that the reader distinguishes which thread has written which

data block and, therefore, the sequence can be lost. Whether or not this is a problem depends on the application. It can be solved by adding a sequence numbers in the data block.

Pipes can also be used without thread synchronization and provide just simple very fast FIFO queues. Those highly optimized functions are very fast. (qPipPut() and qPipGet())

16.2. Using Pipes with Messages

Pipes can be used to transfer messages between threads, fibers and interrupts and provide better synchronization control the queues.

Allocating and de-allocating messages and reading from and writing to a pipe can be done in interrupt service routines because all function operate atomically. This combined with the powerful synchronization control makes them extremely fast to bridge interrupts with the messaging infrastructure.

Pipes handle messages as simple integer wide entities (pointers) and store them in the FIFO queue like any other data element. There is a different function set for messages because the use count is incremented if a message is placed in the pipe and only one message can be read or written per function call.

See messages services for more information.

16.3. Pipe Functions

Function	Description
qPipBlockSize()	Returns the size of the block specified during creation.
qPipClose()	Close an existing Pipe
qPipCreate()	Create a new Pipe
qPipEntries()	Returns the number of entries in the Pipe
qPipGet()	Read one or multiple blocks from the Pipe without calling the notification routine and thread synchronization.
qPipGetFast()	Read one or multiple blocks from the Pipe without calling the notification routine, thread synchronization and parameter checking.
qPipGetBytFast()	Read one byte from the Pipe without calling the notification routine and thread synchronization without parameter checking. (BlockSize must be 1)
qPipGetWrdFast()	Read one word (16 or 32 bit) from the Pipe without calling the notification routine and thread synchronization. (BlockSize must be 2 or 4)
qPipMaxBlocks()	Returns the maximum number of entries (Blocks) the Pipe can hold
qPipOpen()	Return the Pipe by name or wait until a Pipe with that name is created.
qPipOpenNB()	Return the Pipe by name or return null of a with that name does not exists.
qPipOpenTO()	Return the Pipe by name or wait until a Pipe with that name is created with time out.
qPipPut()	Write one or multiple blocks into the Pipe without calling the notification routine and thread synchronization.
qPipPutFast()	Write one or multiple blocks into the Pipe without calling the notification routine, thread synchronization and parameter checking.
qPipPutBytFast()	Write one byte to the Pipe without calling the notification routine and thread synchronization. (BlockSize must be 1)

qPipPutWrdFast()	Write one word (16 or 32 bit) from the Pipe without calling the notification routine and thread synchronization. (BlockSize must be 2 or 4)
qPipRead()	Read one or multiple blocks from a Pipe
qPipReadFast()	Read one blocks from a Pipe without parameter checking
qPipWrite()	Write one or multiple blocks into the Pipe
qPipWriteFast()	Write one blocks into the Pipe without parameter checking

16.4. Pipe Example

See the Publish/Subscribe example.

17. Queue Services

The sole purpose of a queue is to exchange messages between threads and fibers. Multiple messages can reside in a queue. Queues are 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.

Q-Kernel™ queues are fully automatic and require minimal user intervention. Messages are placed in the queue in First In, First Out (FIFO) order. If the queue is full and the sender wants to send, the sender will be blocked until there is space in the queue. If the queue is empty and a receiver tries to receive it will be blocked until the queue contains at least one message. This method is very simple and strait forward.

Queues are very simple to use

17.1. Queue Functions

Function	Description
qQueClose()	Close an open Queue
qQueCreate ()	Create a new Queue
qQueOpen()	Open an existing queue and wait if the queue is not created
qQueOpenNB()	Open an existing queue without blocking
qQueOpenTO()	Open an existing queue and wait if the queue is not created with timeout

17.2. Queue Example

In the following example one thread creates a queue and sends a message. The other thread will receive the message.

```
//-------------------------------------------------
//  Thread code sending thread
//-------------------------------------------------
  pQUE pQue
  pMSG pMsg
  pQue = qSemCreate("Que",6); // Create with max 6
                              // entries
  pMsg = qMsgAlloc(10);       // create a 10 byte mes
  pMsg->Data[0] = mydata1;    // fill some data
    … … … …                   // fill rest of data
  qMsgSend(pQue, pMsg);       // send the message
    … … … …                   // Do other things
  qMsgFree(pMsg);             // Free the message
}

//-------------------------------------------------
//  Thread code receiving thread
//-------------------------------------------------
  pQUE pQue
  pMSG pMsg
  pQue = qSemOpen("Que");     // Open and wait until
                              // queue is available
  pMsg = qMsgReceive(pQue);   // receive message
  if (pMsg->Data[0] == 1) {   // handle data
    … … … …                   // Do other things
  }
  qMsgFree(pMsg);             // Free the message
}
```

18. Publish / subscribe services

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.

- Use the function delivery method to decouple the time domains from publisher and subscriber. This means that messages can be lost. A good example is the measurement every millisecond of a tank level. A display doesn't have to update this level every millisecond so the delivery function just gets the level and updates a global variable. The display is updating the information every second and it just uses the global variable with the value of that time. The publisher (tank-level) and subscriber (display) have different time domains, millisecond versus second, bridged by the pub/sub mechanism.
- Use the pipe delivery method when no information can be lost and the developer needs full control over the information flow. Pipes provide better control to limit thread switching.
- Use the queue delivery method when no information can be lost but the load is limited so full control of the information flow is not required.

Messages are published with the function qMsgPublish() and the payload is a Q▪Kernel™ message (pMSG). 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. Every subscriber will receive this message in a queue, pipe or by a delivery function depending on the delivery method.

- When a message is published and the delivery method is “pipe” Q▪Kernel™ will write the message into the pipe with the function qMsgWrite(). The subscriber can read the message with the function qMsgRead().
- When a message is published and the delivery method is “queue” Q▪Kernel™ will queue the message with the non-blocking function qMsgSendNB() because it can’t preempt. The subscriber can read the message with one of the queue receive functions qMsgReceive(), qMsgReceiveNB() and qMsgReceiveTO().
- When a message is published and the delivery method is a “delivery function” Q▪Kernel™ will call the delivery function with the message as parameter. Depending on the internal state Q▪Kernel™ will call the function in a critical section or as a fiber to prevent collisions. In both cases the delivery function can only use non-blocking functions.

In all cases the subscriber has to free the message.

Subscribers functions (subscribed with qPubSubscribeFun()) are always executed within a critical section and sometimes as a fiber. For that reason blocking functions are not allowed in those functions.

18.1. Pub/sub functions

Function	Description
qPubCreate()	Create a publication based on a name
qPubClose()	Closes a publication and all its subscribers
qPubOpen()	Open a publication and wait until the publication is available
qPubOpenTO()	Open a publication and wait until the publication is available with timeout
qPubOpenNB()	Open a publication without blocking
qPubSubscribeFun()	Subscribe to a publication with a delivery function
qPubSubscribePip()	Subscribe to a publication with a pipe
qPubSubscribeQue()	Subscribe to a publication with a queue
qMsgPublish()	Publish a message to the subscribers.

18.2. Pub/Sub Example

The following example uses a publisher that send messages to 3 subscribers, a function, a queue and a pipe.

```
int main() {
  qKrnInit(……);     // Initialize Q-Kernel
  qThrCreate("Thr1",Thr1,0,256,10); // Function subscr
  qThrCreate("Thr2",Thr2,0,256,20); // Queue subscr
  qThrCreate("Thr3",Thr3,0,256,30); // Pipe subscr
  qThrCreate("Thr9",Thr9,0,256,90); // Publisher
  qKrnStart();      // Never returns here but the…
  return 0;         // …return 0; prevents warnings
}
```

A standard Q-Kernel™ application that creates 4 threads. Thread9 has the highest priority so the start will continue at thread 9.

```
void Thr9(void *p) {
  pPUB pub;
  pMSG msg;
  int n;

  pub = qPubCreate("Pub");  // create the publisher
  qThrSleep(qMSEC(1));      // other threads can start
  for (n=0; n<10000; n++) { // do 10000 times
    msg = qMsgAlloc(2);     // Allocate a message
    msg->Data[0] = n;       // Put counter in data
    qMsgPublish(pub, msg);  // Publish the message
    qMsgFree(msg);          // Free the message
    qThrSleep(45);          // wait 45µSec
  }
}
```

The thread starts with creating the publication. The name is “Pub” which is used by the subscribers. A for loop is used to send the messages. In the loop a message is allocated, a counter is moved into the message and the message is published. The message is freed because this thread is done with it. We wait some time to mimic normal behavior.

As you can see this thread does not have any knowledge of subscribers. This is real loosely coupling and if this thread measured a temperature, every subscriber could use this value. This module exists on its own and has to be tested only once and does not have to change when the application changes. The example is very simple but this could be a complex measurement and/or calculation module that can be re-used without testing.

18.2.1 Subscriber is a function

```
int Counter1 = 0;          // counter for verification

void myFun(void *dummy, pMSG msg) {
  if (Counter1++!=msg->Data[0]) {
    qNtfError(1);          // give error if failure
  }
  qMsgFree(msg);
}

void Thr1(void* p) {
  pPUB pub = qPubOpen("Pub");
  qPubSubscribeFun(pub,myFun);
}
```

Thread 1 is started and is going to open the publication and subscribes to the publication with a function. Because the thread ends, Q·Kernel™ will return all its resources to the resource pools. The function does all the work.

The function will be called when a message is published. The function just checks if the counter matches the expected value but in a real application the function can do useful work. Because the message is managed it looks like this function just owns the message and when done it can free it.

The message can be published by a thread, fiber or ISR. Even if an ISR publishes the message it will never run in an ISR. In all case it will run in a critical section, meaning that no blocking functions (waiting) are allowed.

18.2.2 Subscriber is a queue

A subscriber as queue is the simplest solution to code.

```
void Thr2(void* p) {
  INTU cnt = 0;
  pMSG msg;
  pPUB pub = qPubOpen("Pub");
  pQUE que = qQueCreate(0,10);
  qPubSubscribeQue(pub, que);
  while (1) {
    msg = qMsgReceive(que); // wait until message
    if (cnt++ != msg->Data[0]) {
      qNtfError(2);         // give error if failure
    }
    qMsgFree(msg);
}
```

Thread 2 opens the publication, creates a queue and subscribes the publication to the queue. After initialization it runs an endless loop where it just waits for messages to be received and process the message. Also here it looks like the message is owned by the thread, while in reality it is shared with other threads.

18.2.3 Subscriber is a pipe

A subscriber can use a pipe to execute the messages in the thread but limit thread switching. To do this it waits until the pipe is 80% full. Compared to queues this requires more coding but creates more control over thread switching for a better performance.

```
int Counter2 = 0;          // counter for verification

void myNtfReader(pPIP pip, INTU entriesDone) {
  if (qPipEntries(pip)>=8 || entriesDone==0)
    qThrResume(qThrOpenNB("Thr3"),1);
}

void Thr3(void* p) {
  pPUB pub = qPubOpen("Pub");
  pPIP pip = PipCreate(0,2,10,myNtfReader,0);
  qPubSubscribePip(pub,pip);
  while (1) {
    pMSG msg = qMsgRead(pip);
    if (msg) {
      if (Counter2++!=msg->Data[0]) {
        qNtfError(3);
      }
      qMsgFree(msg);
    }
    else {
      qThrSuspend();
    }
  }
}
```

Thread 3 opens the publication, creates a pipe and subscribes the publication to the pipe. After initialization it runs an endless loop where it tries to read a message. If there is no message it suspends itself and relies on the reader notification function to be resumed.

During the creation of the pipe a read notification function will be defined. This function will be called when something is written in the pipe by the publisher. When there are 8 or more entries in the pipe it will resume the thread. If that is not the case it just returns. This behavior will limit the thread switching and will make the whole system faster. This example prevents 8 context switches.

Also note that the function resumes the thread when entriesDone==0 to handle the case when the pipe will be closed.

19. Message Services

Messages are an essential part of Q-Kernel™ because they can be used to transfer information from threads, fibers and interrupts. While most competitive products implement sending and receiving of messages, Q-Kernel™ supports a much more comprehensive approach to transferring information.

Q-Kernel™ implements several mechanisms for transferring information:

- Queues for simple information exchange
- Pipes for maximum queue control and ISR support
- Publications for loosely coupling of software modules and to promote re-use and agility

All those mechanism require the management of messages. Q-Kernel™ will manage the messages by keeping track of a use count with every message. The use count defines how many software entities, mostly threads, are using the message. After creation, the message count is 1. The use count is incremented if a message is sent to a queue, written in a pipe or publishes to subscribers because it has been made available to others. If a software component de-allocates the message, qMsgFree(), the use-count is decremented. Both sending and receiving threads need to de-allocate the message before the use-count reaches 0 and the object is returned to the memory pool.

The only alternative to managed message is to send messages by value, meaning that messages will be duplicated. This adds a lot of overhead if the messages are large and don't provide any solution for publishing messages because in that case the amount of interested parties in the message is unknown.

The following example describes the simplest case of a sender and a receiver. Without managed messages the priority of the sender and receiver defines who needs to de-allocate the message.

The normal sequence of operation for a sender is as follows:

1. Sender allocates a message
2. Sender fills the message with data
3. Sender sends the message
4. Sender de-allocates the message (if the receiver has a higher priority)

The receiver does the following:

1. Receiver request to receive a message
2. Receiver receives the message
3. Receiver read the data and does whatever it needs to do
4. Receive de-allocates the message (if the sender has a higher priority)

The sender needs to de-allocate the message if the receiver has a higher priority and the receiver needs to de-allocate the message if the send has a higher priority. With Q-Kernel™ both sender and receiver de-allocate the message and this

guarantees that the message is freed-up and the message is returned to the resource pool.

Every thread handles the message as if it totally owns the message and does not have to synchronize with other threads. Q-Kernel™ will handle this transparently.

Message can be handled as if every thread owns the message.

Messages can be allocated from Variable Memory Blocks or Fixed Memory blocks. Messages created from Fixed Memory Blocks can be allocated and de-allocated in Interrupt Service Routines.

- Allocation from Variable Memory is simple. The allocation is based on the required size and the system returns a pointer to the message.
- Allocation from Fixed Memory can be done in interrupt service routines and requires a fixed memory pool address. The function qMsgFixCreate() creates a fixed memory pool for messages and returns a pointer to that pool that can be used later to allocate messages.

A newly allocated message has a use count of one, because it is only used by the unit that allocated it. If a message is sent by queue or pipe the use-count is incremented. This means that the message cannot be written before at least one user of the message, de-allocates the message. If a message use-count is greater than one, others rely on the content of the message and it cannot be written. If one of the users de-allocates the message the other user can now write it. If both users de-allocate the message it is returned to the pool.

The structure that controls the message is MSG. See the structure below:

```
typedef struct sMSG {
  void *Pool;           // DO NOT USE
  uint8_t MemPoolType; // DO NOT USE
  uint8_t UseCnt;       // Use count (READ-ONLY)
  uint16_t MsgType;     // Type of message (READ-ONLY)
  unsigned Data[];      // READ-ONLY if UseCnt > 1
} MSG;                  // READ-WRITE if UseCnt == 1
typedef MSG* pMSG;
```

It must be clear that the Pool and Type can't be used. The other fields have the following meaning:

- UseCnt is read-only and has a value of 1 or higher. If the value is 1 the data can be written. If the value is greater than 1, the data is read-only because others rely on the information in the message.
- The structure defines the Data as a simple array of unsigned integers.

If the array of integers is not practical the developer can define another structure to handle information. The best way to accomplish this is to create a new structure where only the data is a different type. The user has to cast the message to the new format. All structures are supported because Q-Kernel™ guarantees that data is on an integer boundary.

```
// Define the structure
typedef struct sMY_MSG {
  void *Pool;           // DO NOT USE
  uint8_t MemPoolType; // DO NOT USE
  uint8_t UseCnt;       // Use count (READ-ONLY)
  uint16_t MsgType;     // Type of message (READ-ONLY)
  uint8_t Level1;       // READ-ONLY if UseCnt > 1
  uint8_t Level2;       // READ-ONLY if UseCnt > 1
  uint32_t Counter1;    // READ-ONLY if UseCnt > 1
  uint32_t Counter2;    // READ-ONLY if UseCnt > 1
} MY_MSG;               // READ-WRITE if UseCnt == 1
typedef MY_MSG* pMY_MSG;
};
// Define and allocate
pMY_MES mes;
mes = (pMY_MES)qMsgAlloc(sizeof(MY_MES), 10);

// Send the message
qMsgWrite(pPip,(pMSG)mes);
```

19.1. Messages and pipes

Messages can be exchanged by pipes. Pipes are very powerful First In, First Out (FIFO) data structures. Pipes give the user full control over the activation of the reader and writer and are very fast. The combination of pipes and messages is very powerful because both can be managed in interrupt service routines. Messages can be sent to threads, fibers or interrupts. To create a pipe for exchanging message the size of the block (second parameter) should be defined as the size of the pointer like sizeof(pMSG). The third parameter defines the number of blocks which is in this case the number of messages.

Message can be allocated in interrupt handlers and send or received from interrupt handlers

19.2. Messages and queues

Messages can be exchanged by queues. While queues are less powerful than pipes they are much simpler to use. Simple send and receive function are available and threads just preempt when the queue is empty on a receive request or when the queue is full on a send request. Messages can be sent to threads or fibers.

19.3. Messages and publish/subscribe

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 promote re-use and agility. See the Publish/Subscribe Service chapter for more information.

The managing of those messages is very important because the publisher is un-aware how many subscribers are interested in the message and it is possible that this will change. Every time a message is published Q▪Kernel™ will send the message to all subscribers and will keep track of the use count. If a subscriber is done with the message it will “free” the message. If the use-count reaches 0 the message is physically de-allocated.

Managed messages are vital in publish subscribe because the pattern is designed to hide the number of subscribers from every individual subscriber and publisher so it is unknown how the message is used.

19.4. Message Function

Function	Description
qMsgAlloc()	Allocate a message with a specific size.
qMsgCopy()	Allocates a new message and copies a data into the new message.
qMsgFixAlloc()	Allocates a message from a fixed pool.
qMsgFixCreate()	Creates a fixed memory pool for messages.
qMsgFree()	Lower the use count of a message and free the memory if use count is zero
qMsgMaxSize()	Returns the maximum size in bytes that can be stored in the message.
qMsgPublish()	Publish a message to all subscribers.
qMsgRead()	Read a message from a pipe.
qMsgReceive()	Receive a message from a queue and wait if there is no message available
qMsgReceiveNB()	Receive a message from a queue without blocking
qMsgReceiveTO()	Receive a message from a queue and wait with time-out if there is no message available
qMsgSend()	Send a message to a standard queue and wait if there is no room in the queue
qMsgSendNB()	Send a message to a standard queue without blocking
qMsgSendTO()	Send a message to a queue and wait if there is no room in the queue with time-out
qMsgWrite()	Write a message to a pipe.

19.5. Message Example

The following example allocates a message in one thread and sends it to another thread.

The thread does not have to know what the receiving thread does with the message and how long the message is in use.

```
//  Thread code that allocates a message and send it
//  We expect that the queue is already allocated

  pMSG pMsg;                    // The message
  … … … …
  while (… …) {
    INTU size = ……….           // calculate the size
    pMsg = qMsgAlloc(size);     // Allocate it
    for (n=0; n<size; n++) {    // move size integers
      pMsg->Data[n] = data[n];// into the data
    }
    qMsgSend(pQue, pMsg);       // Send the data.
    qMsgFree(pMsg);             // Just free. Managed
                                // message so we don't
                                // have to know what the
                                // receiving thread does
  }                             // with the message
```

```
//  Thread code that receives the message
//  We expect that the queue is already allocated

  while (… …) {
    pMSG pMsg;                  // The message
    pMsg = qMsgReceive(pQue); // Receive the data.
    … … … …                     // Do something with the
    … … … …                     // message. We can use
    … … … …                     // the data without
    … … … …                     // copying
    qMsgFree(pMsg);             // Just free. Managed
                                // message so we don't
                                // have to know what the
                                // sending thread does
  }                             // with the message
```

20. Timer and RTCC Services

A timer object is an object that starts a fiber when the specified due time arrives. Timers are completely disconnected from threads which make them very flexible, and they can be re-used.

There are two types of timers that can be created:

- A One-Shot timer is started and stops when it reaches its specified time.
- A Periodic timer is reactivated each time the specified period expires, until the timer is stopped or closed.

A thread uses the qTmrCreate() function to create a timer object. The creating thread specifies whether the timer is a one-shot timer or a periodic timer, the number of ticks or time, the function to call, the parameter for the function and a name for the timer object. Threads can open an existing timer by specifying its name in a call to the qTmrOpen() function.

The function parameter can be used to exchange information between the creation of the timer and the function. This makes it possible to use one function for multiple timers.

When a timer expires, the system executes the specified function. Some competing products set event flags. This produces 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 switch. The function that will be executed if the time expires can implement that functionality by just signaling an event. Any thread can wait for that event flag. This approach limits the number of context switches and improves performance.

The Q-Kernel timer approach improves performance and facilitates synchronization.

This mechanism allows threads to combine the elapse of a timer with other events. This mechanism can be used if a thread has to synchronize one or more events with periodic time.

Two types of clock sources can be used; the kernel timer or the kernel RTCC. A positive value specifies short times, in cycles implemented by the kernel timer, and a negative value specifies long times, in seconds implemented by the kernel RTCC. The macros qMSEC() and qUSEC() can be used to specify the time in milliseconds or µSeconds using the kernel timer. The qSEC() macro can be used to specify the time in seconds. The accuracy of the time is just as good as the accuracy of the source. The smallest practical time is a few hundred cycles.

20.1. RTCC Services

One of the requirements of a Tick-Less RTOS is the availability of a Real Time Clock and Calendar (RTCC) for long wait times. Q-Kernel™ utilizes timer functions to implement alarm functions. There is an alarm function qRtcAlarm() available for convenience but it is just an interface into the timer functions.

RTCC services are available to set the time (qRtcSetDatTim()) and get the time (qRtcSetDatTim()). The system also provides the uptime in seconds.

20.1.1 Date and Time Formats

The system uses an internal format to store the Date/Time in a 32 bit unsigned integer. The integer contains the number of seconds since Jan, 1st 2000 12:00AM. This format has be chosen to minimize storage and for easy manipulation. For compatibility with other system and for convenient handling of dates and times the system also provides date and time structures in BCD format for simple extraction and display.

The structures are defined as follows:

```
typedef struct sDATETIME {
  uint8_t year;     // BCD codification 0x00->0x99
  uint8_t rsvd;     // Reserved for future use
  uint8_t mday;     // BCD codification 0x01->0x31
  uint8_t mon;      // BCD codification 0x01->0x12
  uint8_t hour;     // BCD codification 0x00->0x23
  uint8_t wday;     // BCD codification 0x00->0x06
  uint8_t min;      // BCD codification 0x00->0x59
  uint8_t sec;      // BCD codification 0x00->0x59
} DATETIME;
typedef DATETIME* pDATETIME;
```

There are functions to convert the internal format to the DATETIME structure.

The system also provides functionality to manipulate dates, like adding or subtracting seconds, minutes, hours, days and years. Every time a date is returned the value "wday" is calculated which specifies the day of the week (0=Sunday, 1=Monday, ... 6=Saturday) for that day.

Q·Kernel provides elaborate date and time functions to minimize development time.

20.2. µSecond Services

Q▪Kernel™ provides a service to calculate time differences accurately. The service counts the number of cycles and calculates[31] the number of µSecond from there. There is no physical counter and no hardware timer is used with the exception of the kernel timer.

This functionality adds only 40 cycles to the processing if the timer expires. Because Q▪Kernel™ is tick-less this occurs very infrequently, so the overhead is very small. While the processing overhead is low, this function can increase power consumption because it keeps the timer running and the power management functionality of Q▪Kernel™ is not able to switch to sleep mode.

The counter starts when the function qKrnUSecOn() will be executed and can be stopped with the function qKrnUSecOff() which clears the counter. Because the function has to synchronize with the timer it will execute a short sleep of a few µSeconds.

There are less granular functions available but the range is more limited. Those functions return a 32-bit unsigned integer and are faster to calculate differences. See the table below:

Function	Size	Granularity	Range
qTimCycles()	64 bit	1 cycle	> 5,000 year
qTimUSec()	64 bit	1 µSec	> 5,000 year
qTimMSec()	64 bit	1 mSec	> 5,000 year

The µSecond timers are in sync with the kernel timer but not with the kernel real-time clock. Their primary purpose is to calculate time differences based on cycles.

The µSecond counter provides a way to calculate time difference accurately.

[31] The calculation is based on shifting and dividing with a 16-bit value so the calculation takes no more than 93 cycles on a 16 bit PIC and less on a PIC32.

20.4. Timer Functions

Function	Description
qTmrClose()	Close an existing timer
qTmrCreate()	Create a new timer
qTmrOpen()	Open an existing timer or wait until the timer is created
qTmrOpenNB()	Open an existing timer without blocking
qTmrOpenTO()	Open an existing timer or wait with timeout until the timer is created
qTmrStart()	Start a timer
qTmrStop()	Stop a timer

20.5. RTCC and Date time Functions

Function	Description
qDtmAddDays()	Add days to a date time and returns the result.
qDtmAddHours()	Add hours to a date time and returns the result.
qDtmAddMinutes()	Add minutes to a date time and returns the result.
qDtmAddSeconds()	Add seconds to a date time and returns the result.
qDtmAddYears()	Add years to a date time and returns the result.
qDtmFromDateTime()	Convert a DATETIME value to the internal format
qDtmFromYMDHMS()	Convert year, month, day, hour, minute and second to the internal format
qDtmToDateTime()	Convert the internal format to a DATETIME value
qRtcAlarm()	Create an alarm function that will be called as a fiber when the due date-time arrives.
qRtcSetDatTim()	Set the current date and time
qRtcGetDatTim()	Returns the current date and time if set.
qRtcGetUptime()	Returns the number of seconds since startup

21. Installing and using Q▪Kernel™

Q▪Kernel™ is distributed as a zip file with the name qKernelV3363.zip. The version contains all versions of Q▪Kernel™ in that build. The last 4 digits in the name is the build number. Extract the file into a main directory like C:\qKernelFree, but it can be any drive or name because all references are relative. An example of the directory structure is specified below.

```
V3353
 ----- Documentation
 ----- Source
 ----- ------Pic24_MPLAB.X (PIC24 /dsPIC port)
 ----- ------ -----Blinky.X (Test program)
 ----- ------ ----- ----- dist
 ----- ------ ----- ----- ------default
 ----- ------ ----- ----- ------------production
 ----- ------ ----- ----- nbproject
 ----- ------ -----dist (Distribution for the Q▪Kernel™ libraries)
 ----- ------ ----- ----- default
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- Generic
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- GenericDA
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- GenericEP
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- ThreadMetric
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ -----nbproject
 ----- ------ -----ThreadMetric.X
 ----- ------ ----- ----- dist
 ----- ------ ----- ----- ------default
 ----- ------ ----- ----- ------------production
 ----- ------ ----- ----- nbproject
 ----- ------ ----- ----- ------private
 ----- ------Pic32_MPLAB.X (PIC32 port)
 ----- ------ -----Blinky.X (Test program)
 ----- ------ ----- ----- dist
 ----- ------ ----- ----- ------default
 ----- ------ ----- ----- ------------production
 ----- ------ ----- ----- nbproject
 ----- ------ -----dist (Distribution for the Q▪Kernel™ libraries)
 ----- ------ ----- ----- default
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- Generic
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ -----nbproject
```

21.1. Adding Q-Kernel™ to your application

Q-Kernel™ can be included as a project library or as an object library. We recommend to use the object library because this prevents building the library every time.

Your application need to include “qKernel.h” in all source files in your project. The best way to do that is to add the path to this file in the “C include dirs.” of the compiler.

21.1.1 Using an object library

As described above you can add Q-Kernel™ as an object library. Include one of the standard configurations that is included in the distribution. The libraries are compiled with -s as optimization level, kernel timer is TMR4/TMR5 and the full parameter checking.

21.1.2 Using a project library

You create your application and include Q-Kernel™ as a library project. You go to properties of your project and click libraries. There you click “Add library Project” and select qKernel.X. In this case the source will be included and MPLAB will compile the source if required. You have to specify the configuration, for things like MCU, optimization level, etc.

Q-Kernel™ consists of many files so only the code that is required will be in flash. This makes building slow and even if a full build executes sporadic it is advisable to use the full computer potential. You can improve the build time by selecting Options from the main menu, then Embedded and select the tab “Project Options” Select the option “Use parallel make” to speed up the build. It will use all your cores of your processors.

Improve the compile speed by selecting the option “Use parallel make” and use all the available cores

You can use all other compile and build options, like memory model, optimization levels, etc. for your project or Q-Kernel™ so it is very flexible.

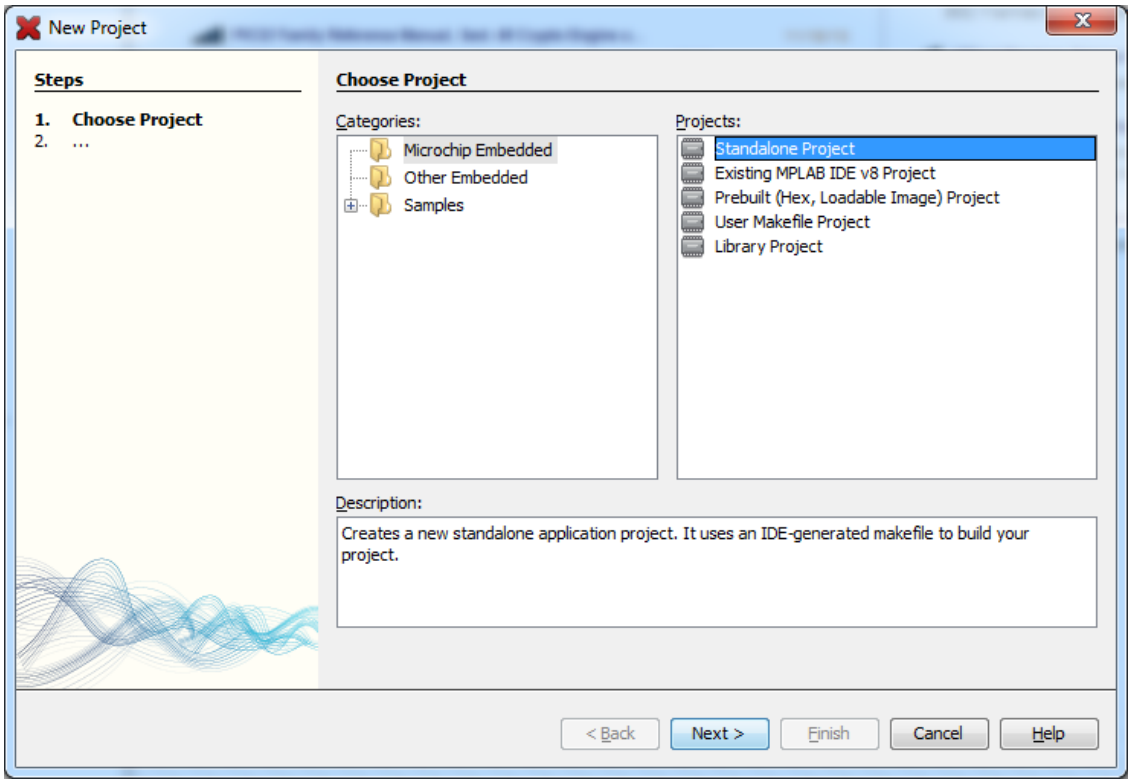
21.2. Using your own object library

First create a configuration within the Q-Kernel™ distribution and set your compile options. Build the system and include the created object library in your application.

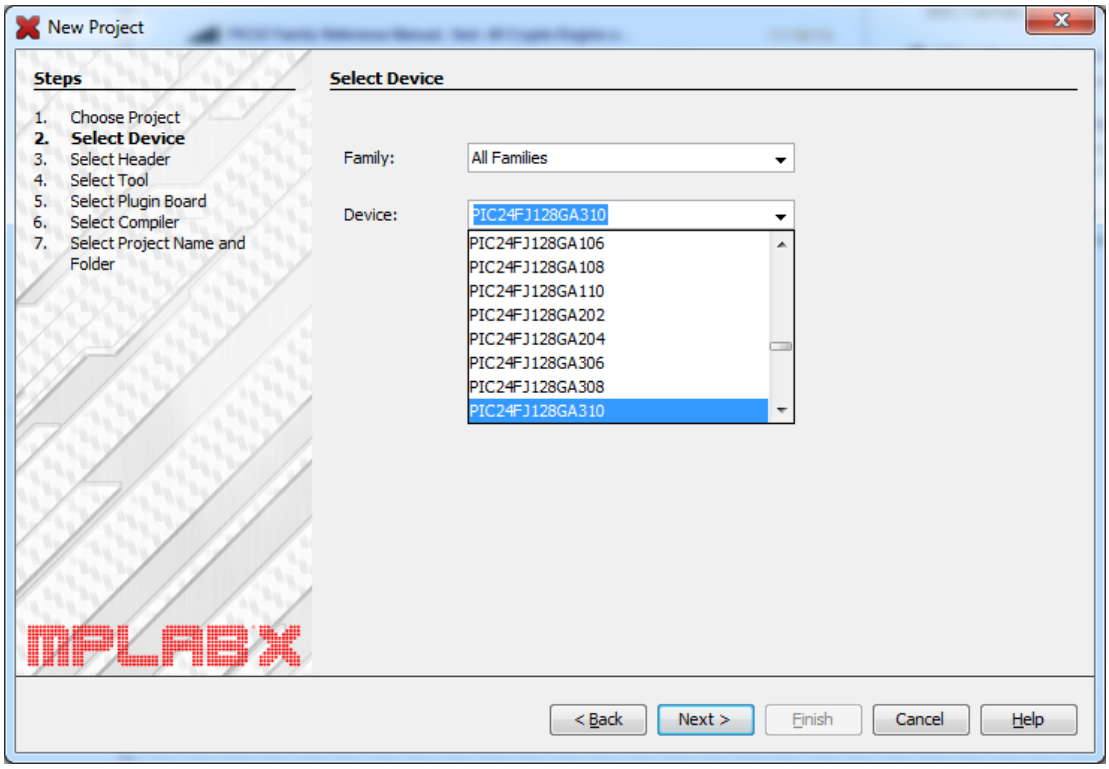
21.3. Creating a new application

This example creates a simple application. While it looks complicated the whole thing is very simple. See the following screen shots.

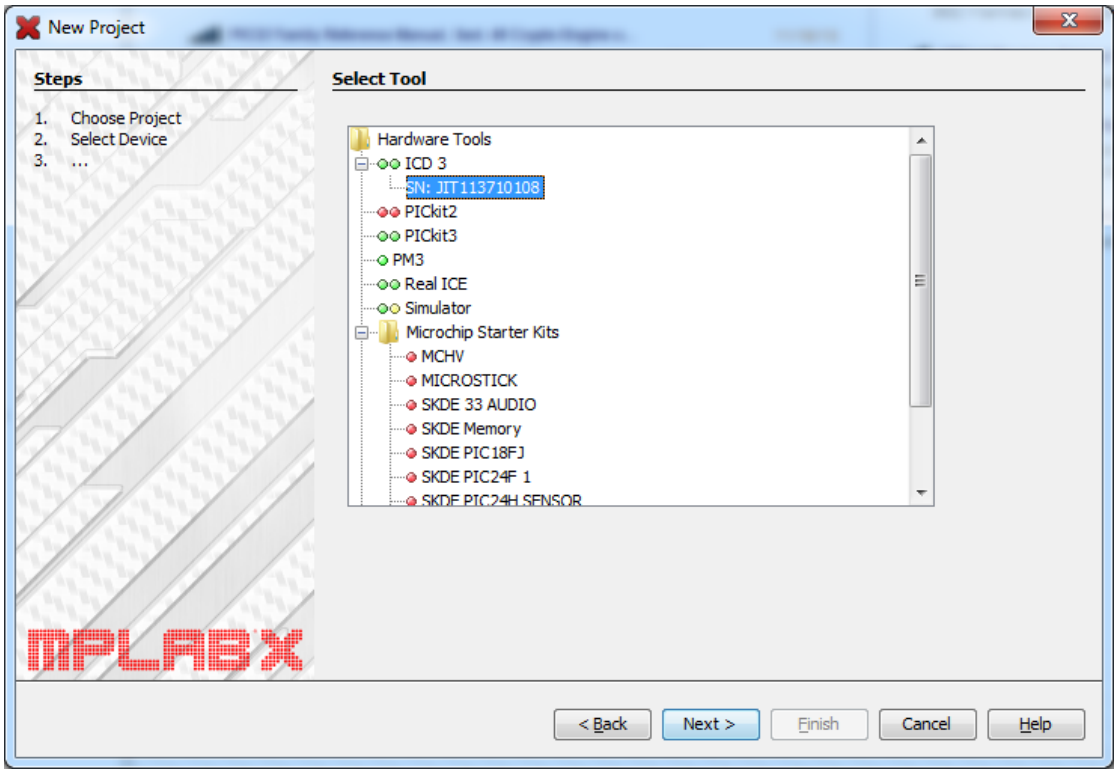
Start a new project



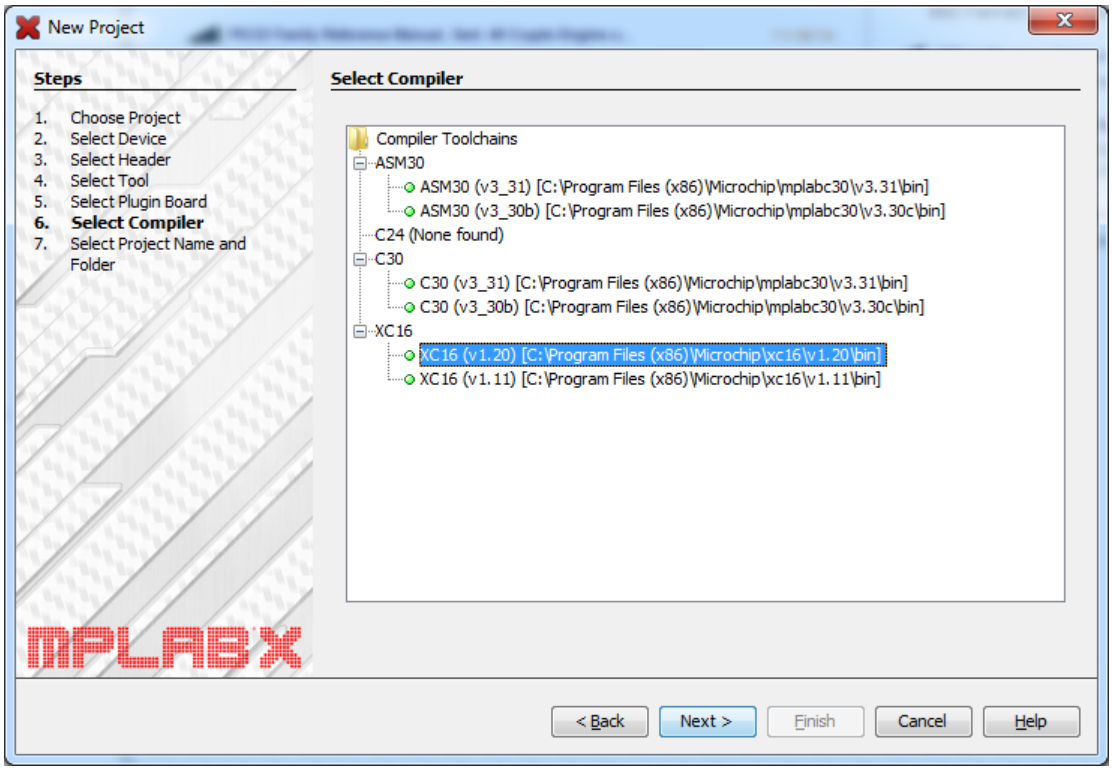
Select the processor



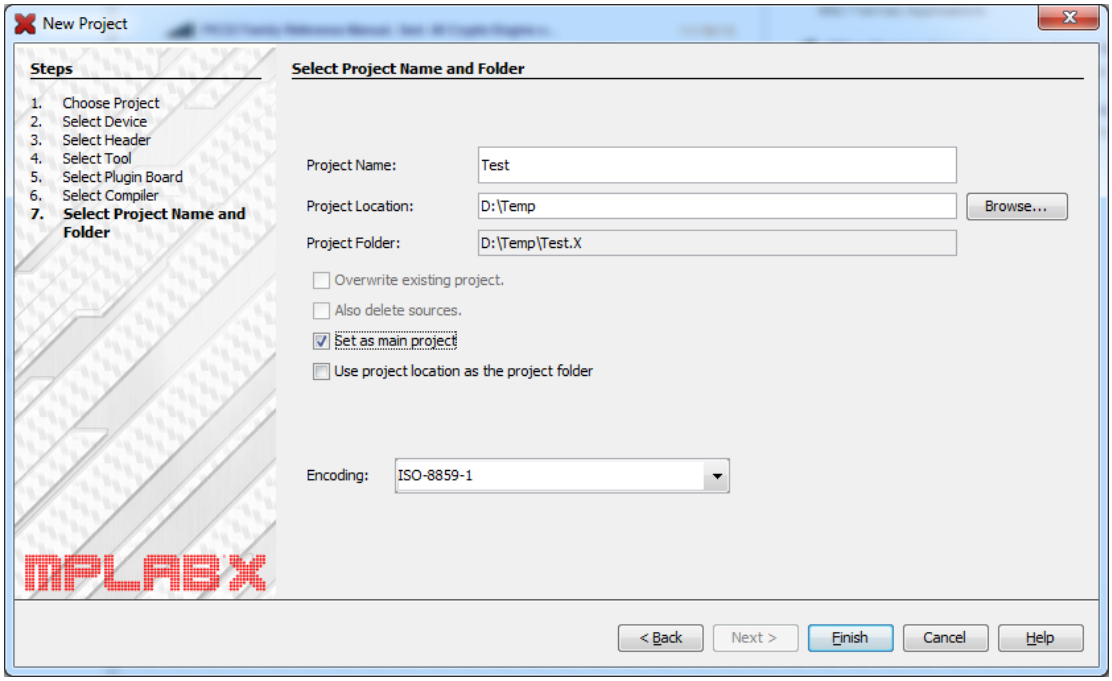
Select the debugging tool (ICD3)



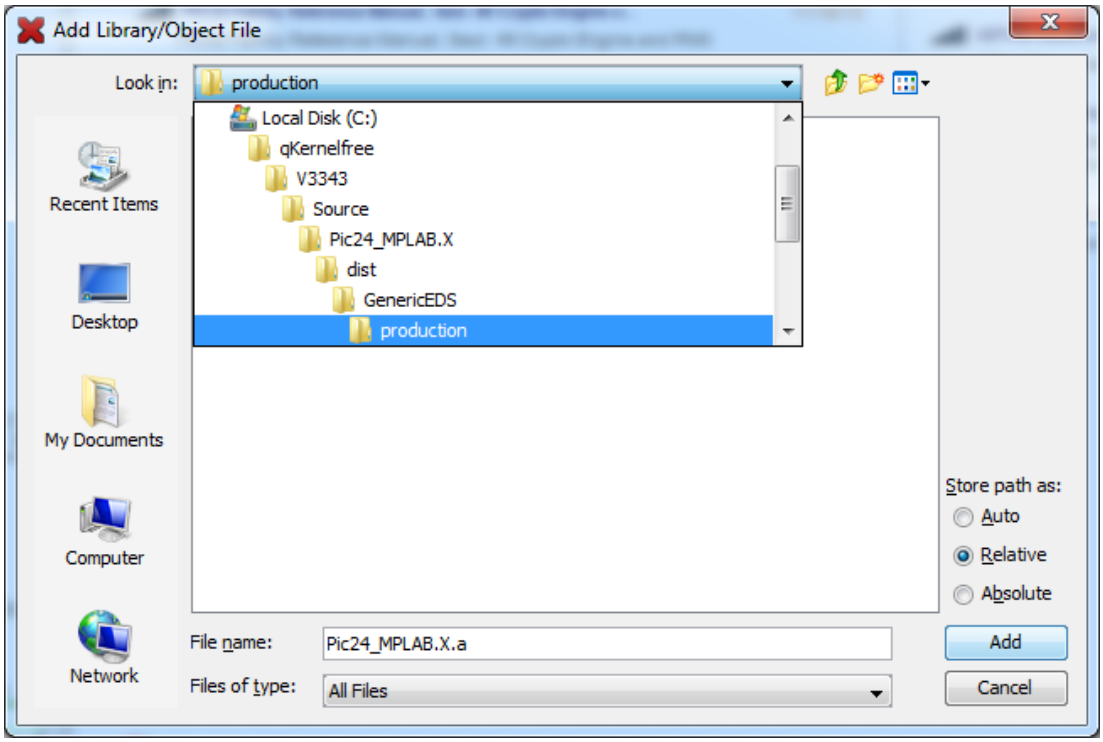
Select a compiler



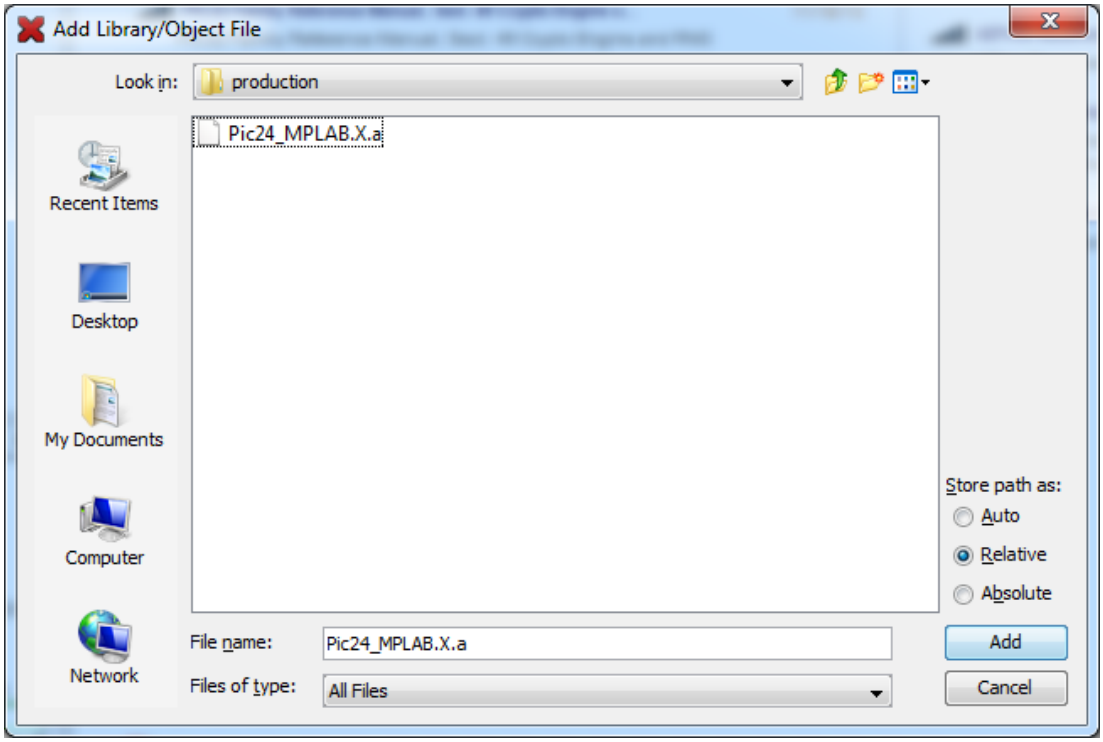
And the project name and folder (See that we use the D: drive)



Add an object library directory (We use GenericEDS because that's our processor)



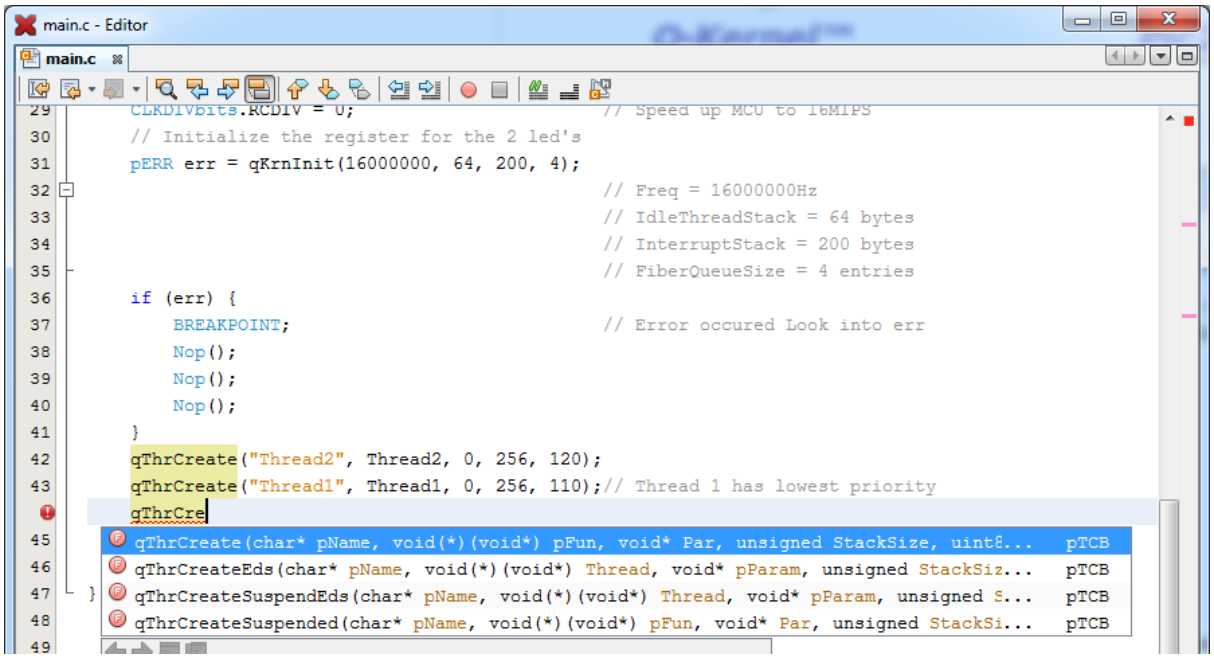
And select the file itself by pressing the Add button



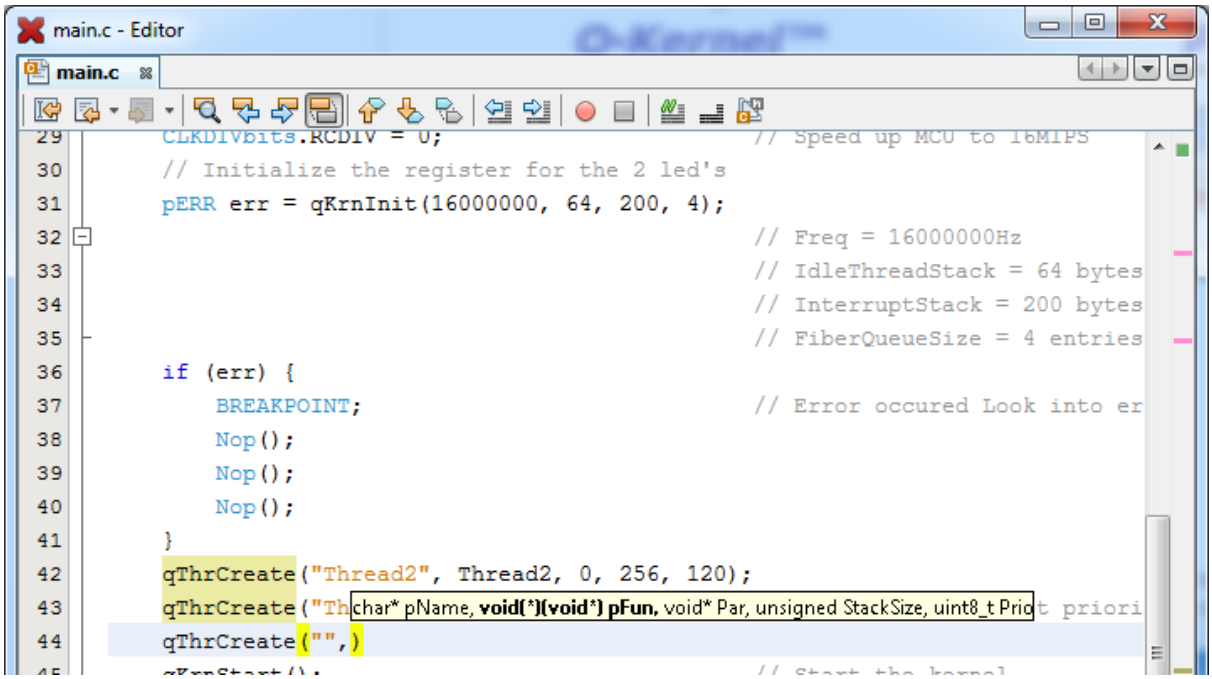
Now create a main c file and add some code like this example (See blinky)

```
#include <stdio.h>
#include <stdlib.h>
#include "C:\qKernelfree\V3343\Source\Pic24_MPLAB.X\qKernel.h"

void Thread1(void* Dummy) {
    while (1) {
        //Toggle led here
        qThrSleep(300000);
    }
}

void Thread2(void* Dummy) {
    while (1) {
        //Toggle (another) led here
        qThrSleep(100000);
    }
}

unsigned qErrNotify(unsigned Error) {           // Central error handling
    if (Error==0) {                             // If error = 0 just return
        return 0;
    }
    qKrnError(Error);                           // This will reset the system..
    return Error;                               // ..and trigger breakpoint
}


int main(int argc, char** argv) {
    CLKDIVbits.RCDIV = 0;                       // Speed up MCU to 16MIPS
    // Initialize the register for the 2 led's
    pERR err = qKrnInit(16000000, 64, 200, 4);
                                                // Freq = 16000000Hz
                                                // IdleThreadStack = 64 bytes
                                                // InterruptStack = 200 bytes
                                                // FiberQueueSize = 4 entries
    if (err) {
        BREAKPOINT;                             // Error occured Look into err
        Nop();
        Nop();
        Nop();
    }
    qThrCreate("Thread2", Thread2, 0, 256, 120);
    qThrCreate("Thread1", Thread1, 0, 256, 110);// Thread 1 has lowest priority
    qKrnStart();                                // Start the kernel
    return 0;                                   // We should never get here
}
```

During typing you can use <ctrl><space> to find the function name.



During typing MPLAB helps you with the parameters



Some remarks:

1. The #include statement (line 3) in the program needs to find file qKernel.h in the porting directory. In the Blinky.X example we use a relative specification because both Q▪Kernel™ and the project are on the C: drive. Use the “preprocessing and messages” dialog and specify the “C include dirs.”
2. Even if Q▪Kernel™ is included as an object library (versus a project) the debugger will find all sources related to the object and the cursor (green line) will show the Q▪Kernel™ code.

22. Glossary

Active Thread	Only one thread can execute at any given time. The thread that is currently executing is called the active thread.
Cooperative multi-threading	A scheduling system in which each thread is allowed to run until it gives up the CPU; an ISR can make a higher priority thread ready, but the interrupted thread will be returned to and finished first.
CPU.	The Central Processing Unit is the "brain" of a microcontroller; the part of a processor that carries out instructions.
Critical section	A section of code which must be executed as a whole and can't be interrupted by the scheduler. **It can be interrupted by Interrupt Service Routines.**
Event	A message sent to a single, specified thread that something has occurred. The thread then becomes ready.
Fiber	A program running in a cooperative multitasking environment. Fibers yield themselves to run another fiber while executing. Fibers don't need a context switch to be activated.
ISR Interrupt Service Routine	The routine is called automatically by the processor when an interrupt is acknowledged. ISRs must preserve the entire context of a thread (all registers).
Message Queue	A data buffer managed by the RTOS, used for sending and receiving messages to and from a thread.
Message	An item of data sent to or received from a Message Queue.
Multi-threading	The execution of multiple software routines independently of one another. The RTOS divides the processor's time so that the different routines (threads) appear to be happening simultaneously.
Preemptive multi-threading	A scheduling system in which the highest priority thread that is ready will always be executed.
Priority (Thread)	The relative importance of one thread to another. Every thread in an RTOS has a priority.

Priority inversion	A condition in which a high priority thread is delayed while it waits for access to a shared resource which is in use by a lower priority thread. The lower priority thread temporarily gets the highest priority until it releases the resource.
Queue	A data structure for sending and receiving character or integer wide data between threads or between threads and ISR
Resource	Anything in the computer system with limited availability (e.g. memory, timers, computation time). Essentially, anything used by a thread.
RTOS Real-time Operating System.	The program section of an RTOS that selects the active thread, based on which threads are ready to run, their relative priorities, and the scheduling system being used.
Semaphore	A data structure that keeps track of multiple resources. Used when a thread must wait for something that can be signaled more than once.
Software timer	A data structure which calls a user-specified routine after a specified delay.
Stack	An area of memory with FIFO storage of parameters, automatic variables, returns addresses, and other information that needs to be maintained across function calls. In multi-threading systems, each thread normally has its own stack.
Superloop	A program that runs in an infinite loop and uses no real-time kernel. ISRs are used for real-time parts of the software.
Thread	A program running on a processor. A multi-threading system allows multiple threads to execute independently from one another.
Tick	The OS timer interrupt. Usually equals 1 millisecond.
Tick-less	A RTOS that does not use a tick is called a tick-less RTOS. Tick-less improves granularity and lowers power consumption.