Mark3 Realtime Kernel

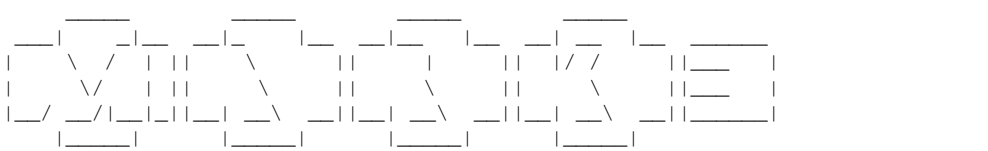
Generated by Doxygen 1.8.6

Thu Mar 5 2015 22:35:15

Contents

Chapter 1

The Mark3 Realtime Kernel



```
--[Mark3 Realtime Platform]----------------------------------------------
```


The Mark3 Realtime Kernel is a completely free, open-source, real-time operating system aimed at bringing multi-tasking to microcontroller systems without MMUs.

It uses modern programming languages and concepts (it's written entirely in C++) to minimize code duplication, and its object-oriented design enhances readibility. The API is simple - there are only six functions required to set up the kernel, initialize threads, and start the scheduler.

The source is fully-documented with example code provided to illustrate concepts. The result is a performant RTOS, which is easy to read, easy to understand, and easy to extend to fit your needs.

But Mark3 is bigger than just a real-time kernel, it also contains a number of class-leading features:

- Device driver HAL which provides a meaningful abstraction around device-specific peripherals.
- Capable recursive-make driven build system which can be used to build all libraries, examples, tests, documentation, and user-projects for any number of targets from the command-line.
- Graphics and UI code designed to simplify the implementation of systems using displays, keypads, joysticks, and touchscreens
- Standards-based custom communications protocol used to simplify the creation of host tools
- A bulletproof, well-documented bootloader for AVR microcontrollers
- Support for kernel-aware simulators, specifically, Funkenstein Software's own flAVR AVR simulator

Chapter 2

Preface

2.1 Who should read this

As the cover clearly states, this is a book about the Mark3 real-time kernel. I assume that if you're reading this book you have an interest in some, if not all, of the following subjects:

- Embedded systems
- Real-time systems
- Operating system kernel design

And if you're interested in those topics, you're likely familiar with C and C++ and the more you know, the easier you'll find this book to read. And if C++ scares you, and you don't like embedded, real-time systems, you're probably looking for another book. If you're unfamiliar with RTOS fundamentals, I highly suggest searching through the vast amount of RTOS-related articles on the internet to familiarize yourself with the concepts.

2.2 Why Mark3?

My first job after graduating from university in 2005 was with a small company that had a very old-school, low-budget philosophy when it came to software development. Every make-or-buy decision ended with "make" when it came to tools. It was the kind of environment where vendors cost us money, but manpower was free. In retrospect, we didn't have a ton of business during the time that I worked there, and that may have had something to do with the fact that we were constantly short on ready cash for things we could code ourselves.

Early on, I asked why we didn't use industry-standard tools - like JTAG debuggers or IDEs. One senior engineer scoffed that debuggers were tools for wimps - and something that a good programmer should be able to do without. After all - we had serial ports, GPIOs, and a bi-color LED on our boards. Since these were built into the hardware, they didn't cost us a thing. We also had a single software "build" server that took 5 minutes to build a 32k binary on its best days, so when we had to debug code, it was a painful process of trial and error, with lots of Youtube between iterations. We complained that tens of thousands of dollars of productivity was being flushed away that could have been solved by implementing a proper build server - and while we eventually got our wish, it took far more time than it should have.

Needless to say, software development was painful at that company. We made life hard on ourselves purely out of pride, and for the right to say that we walked "up-hills both ways through 3 feet of snow, everyday". Our code was tied ever-so-tightly to our hardware platform, and the system code was indistinguishable from the application. While we didn't use an RTOS, we had effectively implemented a 3-priority threading scheme using a carefully designed interrupt nesting scheme with event flags and a while(1) superloop running as a background thread. Nothing was abstracted, and the code was always optimized for the platform, presumably in an effort to save on code size and wasted cycles. I asked why we didn't use an RTOS in any of our systems and received dismissive scoffs - the overhead from thread switching and maintaining multiple threads could not be tolerated in our systems according

to our chief engineers. In retrospect, our ad-hoc system was likely as large as my smallest kernel, and had just as much context switching (althrough it was hidden by the compiler).

And every time a new iteration of our product was developed, the firmware took far too long to bring up, because the algorithms and data structures had to be re-tooled to work with the peripherals and sensors attached to the new boards. We worked very hard in an attempt to reinvent the wheel, all in the name of producing "efficient" code.

Regardless, I learned a lot about software development.

Most important, I learned that good design is the key to good software; and good design doesn't have to come at a price. In all but the smallest of projects, the well-designed, well-abstracted code is not only more portable, but it's usually smaller, easier to read, and easier to reuse.

Also, since we had all the time in the world to invest in developing our own tools, I gained a lot of experience building them, and making use of good, free PC tools that could be used to develop and debug a large portion of our code. I ended up writing PC-based device and peripheral simulators, state-machine frameworks, and abstractions for our horrible ad-hoc system code. At the end of the day, I had developed enough tools that I could solve a lot of our development problems without having to re-inventing the wheel at each turn. Gaining a background in how these tools worked gave me a better understanding of how to use them - making me more productive at the jobs that I've had since.

I am convinced that designing good software takes honest effort up-front, and that good application code cannot be written unless it is based on a solid framework. Just as the wise man builds his house on rocks, and not on sand, wise developers write applications based on a well-defined platforms. And while you can probably build a house using nothing but a hammer and sheer will, you can certainly build one a lot faster with all the right tools.

This conviction lead me to development my first RTOS kernel in 2009 - FunkOS. It is a small, yet surprisingly full-featured kernel. It has all the basics (semaphores, mutexes, round-robin and preemptive scheduling), and some pretty advanced features as well (device drivers and other middleware). However, it had two major problems - it doesn't scale well, and it doesn't support many devices.

While I had modest success with this kernel (it has been featured on some blogs, and still gets around 125 downloads a month), it was nothing like the success of other RTOS kernels like uC/OS-II and FreeRTOS. To be honest, as a one-man show, I just don't have the resources to support all of the devices, toolchains, and evaluation boards that a real vendor can. I had never expected my kernel to compete with the likes of them, and I don't expect Mark3 to change the embedded landscape either.

My main goal with Mark3 was to solve the technical shortfalls in the FunkOS kernel by applying my experience in kernel development. As a result, Mark3 is better than FunkOS in almost every way; it scales better, has lower interrupt latency, and is generally more thoughtfully designed (all at a small cost to code size).

Another goal I had was to create something easy to understand, that could be documented and serve as a good introduction to RTOS kernel design. The end result of these goals is the kernel as presented in this book - a full source listing of a working OS kernel, with each module completely documented and explained in detail.

Finally, I wanted to prove that a kernel written entirely in C++ could perform just as well as one written in C, without incurring any extra overhead. Comparing the same configuration of Mark2 to Mark3, the code size is remarkably similar, and the execution performance is just as good. Not only that, but there are fewer lines of code. The code is more readable and easier to understand as a result of making use of object-oriented concepts provided by C++. Applications are easier to write because common concepts are encapsulated into objects (Threads, Semaphores, Mutexes, etc.) with their own methods and data, as opposed to APIs which rely on lots of explicit pointer-passing, type casting, and other operations that are typically considered "unsafe" or "advaned topics" in C.

Chapter 3

Can you Afford an RTOS?

Of course, since you're reading the manual for an RTOS that I've been developing for the last few years, you can guess that the conclusion that I draw is a resounding "yes".

If your code is of any sort of non-trivial complexity (say, at least a few-thousand lines), then a more appropriate question would be "can you afford ∗not∗ to use an RTOS in your system?".

In short, there are simply too many benefits of an RTOS to ignore.

- Sophisticated synchronization objects
- The ability to efficiently block and wait
- Enhanced responsiveness for high-priority tasks
- Built in timers
- Built in efficient memory management

Sure, these features have a cost in code space and RAM, but from my experience the cost of trying to code around a lack of these features will cost you as much - if not more. The results are often far less maintainable, error prone, and complex. And that simply adds time and cost. Real developers ship, and the RTOS is quickly becoming one of the standard tools that help keep developers shipping.

3.1 Intro

(Note - this article was written for the C-based Mark2 kernel, which is slightly different. While the general principles are the same, the numbers are not an 100% accurate reflection of the current costs of the Mark3 kernel.)

One of the main arguments against using an RTOS in an embedded project is that the overhead incurred is too great to be justified. Concerns over "wasted" RAM caused by using multiple stacks, added CPU utilization, and the "large" code footprint from the kernel cause a large number of developers to shun using a preemptive RTOS, instead favoring a non-preemptive, application-specific solution.

I believe that not only is the impact negligible in most cases, but that the benefits of writing an application with an RTOS can lead to savings around the board (code size, quality, reliability, and development time). While these other benefits provide the most compelling case for using an RTOS, they are far more challenging to demonstrate in a quantitative way, and are clearly documented in numerous industry-based case studies.

While there is some overhead associated with an RTOS, the typical arguments are largely unfounded when an RTOS is correctly implemented in a system. By measuring the true overhead of a preemptive RTOS in a typical application, we will demonstrate that the impact to code space, RAM, and CPU usage is minimal, and indeed acceptable for a wide range of CPU targets.

To illustrate just how little an RTOS impacts the size of an embedded software design we will look at a typical microcontroller project and analyze the various types of overhead associated with using a pre-emptive realtime kernel versus a similar non-preemptive event-based framework.

RTOS overhead can be broken into three distinct areas:

- Code space: The amount of code space eaten up by the kernel (static)
- Memory overhead: The RAM associated wtih running the kernel and application threads.
- Runtime overhead: The CPU cycles required for the kernel's functionality (primarily scheduling and thread switching)

While there are other notable reasons to include or avoid the use of an RTOS in certain applications (determinism, responsiveness, and interrupt latency among others), these are not considered in this discussion - as they are difficult to consider for the scope of our "canned" application. Application description:

For the purpose of this comparison, we first create an application using the standard preemptive Mark3 kernel with 2 system threads running: A foreground thread and a background thread. This gives three total priority levels in the system - the interrupt level (high), and two application priority threads (medium and low), which is quite a common paradigm for microcontroller firmware designs. The foreground thread processes a variety of time-critical events at a fixed frequency, while the background thread processes lower priority, aperiodic events. When there are no background thread events to process, the processor enters its low-power mode until the next interrupt is acknowledged.

The contents of the threads themselves are unimportant for this comparison, but we can assume they perform a variety of I/O using various user-input devices and a serial graphics display. As a result, a number of Mark3 device drivers are also implemented.

The application is compiled for an ATMega328p processor which contains 32kB of code space in flash, and 2kB of RAM, which is a lower-mid-range microcontroller in Atmel's 8-bit AVR line of microcontrollers. Using the WinAVR GCC compiler with -O2 level optimizations, an executable is produced with the following code/RAM utilization:

31600 Bytes Code Space 2014 Bytes RAM

An alternate version of this project is created using a custom "super-loop" kernel, which uses a single application thread and provides 2 levels of priority (interrupt and application). In this case, the event handler processes the different priority application events to completion from highest to lowest priority.

This approach leaves the application itself largely unchanged. Using the same optimization levels as the preemptive kernel, the code compiles as follows:

29904 Bytes Code Space 1648 Bytes RAM

3.2 Memory overhead:

At first glance, the difference in RAM utilization seems quite a lot higher for the preemptive mode version of the application, but the raw numbers don't tell the whole story.

The first issue is that the cooperative-mode total does not take into account the system stack - whereas these values are included in the totals for RTOS version of the project. As a result, some further analysis is required to determine how the stack sizes truly compare.

In cooperative mode, there is only one thread of execution - so considering that multiple event handlers are executed in turn, the stack requirements for cooperative mode is simply determined by those of the most stack-intensive event handler.

In contrast, the preemptive kernel requires a separate stack for each active thread, and as a result the stack usage of the system is the sum of the stacks for all threads.

Since the application and idle events are the same for both preemptive and cooperative mode, we know that their (independent) stack requirements will be the same in both cases.

For cooperative mode, we see that the idle thread stack utilization is lower than that of the application thread, and so the application thread's determines the stack size requirement. Again, with the preemptive kernel the stack utilization is the sum of the stacks defined for both threads.

As a result, the difference in overhead between the two cases becomes the extra stack required for the idle thread - which in our case is (a somewhat generous) 64 bytes.

The numbers still don't add up completely, but looking into the linker output we see that the rest of the difference comes from the extra data structures used to declare the threads in preemptive mode.

With this taken into account, the true memory cost of a 2-thread system ends up being around 150 bytes of RAM - which is less than 8% of the total memory available on this particular microcontroller. Whether or not this is reasonable certainly depends on the application, but more importantly, it is not so unreasonable as to eliminate an RTOS-based solution from being considered.

3.3 Code Space Overhead:

The difference in code space overhead between the preemptive and cooperative mode solutions is less of an issue. Part of this reason is that both the preemptive and cooperative kernels are relatively small, and even an average target device (like the Atmega328 we've chosen) has plenty of room.

Mark3 can be configured so that only features necessary for the application are included in the RTOS - you only pay for the parts of the system that you use. In this way, we can measure the overhead on a feature-by-feature basis, which is shown below for the kernel as configured for this application:

3466 Bytes

The configuration tested in this comparison uses the thread/port module with timers, drivers, and semaphores, for a total kernel size of $\sim$3.5KB, with the rest of the code space occupied by the application.

The custom cooperative-mode framework has a similar structure which is broken down by module as follows:

1850 Bytes

As can be seen from the compiler's output, the difference in code space between the two versions of the application is about 1.7kB - or about 5% of the available code space on the selected processor. While nearly all of this comes from the added overhead of the kernel, the rest of the difference comes the changes to the application necessary to facilitate the different frameworks.

3.4 Runtime Overhead

On the cooperative kernel, the overhead associated with running the thread is the time it takes the kernel to notice a pending event flag and launch the appropriate event handler, plus the timer interrupt execution time.

Similarly, on the preemptive kernel, the overhead is the time it takes to switch contexts to the application thread, plus the timer interrupt execution time.

The timer interrupt overhead is similar for both cases, so the overhead then becomes the difference between the following:

Preemptive mode:

- Posting the semaphore that wakes the high-priority thread
- Performing a context switch to the high-priority thread

Cooperative mode:

- Setting the high-priority thread's event flag
- Acknowledging the event from the event loop

Using the cycle-accurate AVR simulator, we find the end-to-end event sequence time to be 20.4us for the cooperative mode scheduler and 44.2us for the preemptive, giving a difference of 23.8us.

With a fixed high-priority event frequency of 33Hz, we achieve a runtime overhead of 983.4us per second, or 0.-0983% of the total available CPU time. Now, obviously this value would expand at higher event frequencies and/or slower CPU frequencies, but for this typical application we find the difference in runtime overhead to be neglible for a preemptive system. Analysis:

For the selected test application and platform, including a preemptive RTOS is entirely reasonable, as the costs are low relative to a non-preemptive kernel solution. But these costs scale relative to the speed, memory and code space of the target processor. Because of these variables, there is no "magic bullet" environment suitable for every application, but Mark3 attempts to provide a framework suitable for a wide range of targets.

On the one hand, if these tests had been performed on a higher-end microcontroller such as the ATMega1284p (containing 128kB of code space and 16kB of RAM), the overhead would be in the noise. For this type of resource-rich microcontroller, there would be no reason to avoid using the Mark3 preemptive kernel.

Conversely, using a lower-end microcontroller like an ATMega88pa (which has only 8kB of code space and 1k-B of RAM), the added overhead would likely be prohibitive for including a preemptive kernel. In this case, the cooperative-mode kernel would be a better choice.

As a rule of thumb, if one budgets 10% of a microcontroller's code space/RAM for a preemptive kernel's overhead, you should only require at minimum a microcontroller with 16k of code space and 2kB of RAM as a base platform for an RTOS. Unless there are serious constraints on the system that require much better latency or responsiveness than can be achieved with RTOS overhead, almost any modern platform is sufficient for hosting a kernel. In the event you find yourself with a microprocessor with external memory, there should be no reason to avoid using an RTOS at all.

Chapter 4

Superloops

4.1 Intro to Superloops

Before we start taking a look at designing a real-time operating system, it's worthwhile taking a look through one of the most-common design patterns that developers use to manage task execution in embedded systems - Superloops.

Systems based on superloops favor the system control logic baked directly into the application code, usually under the guise of simplicity, or memory (code and RAM) efficiency. For simple systems, superloops can definitely get the job done. However, they have some serious limitations, and are not suitable for every kind of project. In a lot of cases you can squeak by using superloops - especially in extremely constrained systems, but in general they are not a solid basis for reusable, portable code.

Nonetheless, a variety of examples are presented here- from the extremely simple, to cooperative and liimted-preemptive multitasking systems, all of which are examples are representative of real-world systems that I've either written the firmware for, or have seen in my experience.

4.2 The simplest loop

Let's start with the simplest embedded system design possible - an infinite loop that performs a single task repeatedly:

```
int main()
{
    while(1)
    {
        Do_Something();
    }
}
```

Here, the code inside the loop will run a single function forever and ever. Not much to it, is there? But you might be surprised at just how much embedded system firmware is implemented using essentially the same mechanism - there isn't anything wrong with that, but it's just not that interesting.

While the execution timeline for this program is equally boring, for the sake of completeness it would look like this:

Despite its simplicity we can see the beginnings of some core OS concepts. Here, the while(1) statement can be logically seen as the he operating system kernel - this one control statement determines what tasks can run in the system, and defines the constraints that could modify their execution. But at the end of the day, that's a big part of what a kernel is - a mechanism that controls the execution of application code.

The second concept here is the task. This is application code provided by the user to perform some useful purpose in a system. In this case Do_something() represents that task - it could be monitoring blood pressure, reading a sensor and writing its data to a terminal, or playing an MP3; anything you can think of for an embedded system to do. A simple round-robin multi-tasking system can be built off of this example by simply adding additional tasks in

sequence in the main while-loop. Note that in this example the CPU is always busy running tasks - at no time is the CPU idle, meaning that it is likely burning a lot of power.

While we conceptually have two separate pieces of code involved here (an operating system kernel and a set of running tasks), they are not logically separate. The OS code is indistinguishable from the application. It's like a single-celled organism - everything is crammed together within the walls of an indivisible unit; and specialized to perform its given function relying solely on instinct.

4.3 Interrupt-Driven Super-loop

In the previous example, we had a system without any way to control the execution of the task- it just runs forever. There's no way to control when the task can (or more importantly can't) run, which greatly limits the usefulness of the system. Say you only want your task to run every 100 miliseconds - in the previous code, you have to add a hard-coded delay at the end of your task's execution to ensure your code runs only when it should.

Fortunately, there is a much more elegant way to do this. In this example, we introduce the concept of the synchronization object. A Synchronization object is some data structure which works within the bounds of the operating system to tell tasks when they can run, and in many cases includes special data unique to the synchronization event. There are a whole family of synchronization objects, which we'll get into later. In this example, we make use of the simplest synchronization primitive - the global flag.

With the addition of synchronization brings the addition of event-driven systems. If you're programming a microcontroller system, you generally have scores of peripherals available to you - timers, GPIOs, ADCs, UARTs, ethernet, USB, etc. All of which can be configured to provide a stimulus to your system by means of interrupts. This stimulus gives us the ability not only to program our micros to do_something(), but to do_something() if-and-only-if a corresponding trigger has occurred.

The following concepts are shown in the example below:

```
volatile K_BOOL something_to_do = false;

__interrupt__ My_Interrupt_Source(void)
{
    something_to_do = true;
}

int main()
{
    while(1)
    {
        if( something_to_do )
        {
            Do_something();
            something_to_do = false;
        }
        else
        {
            Idle();
        }
    }
}
```

So there you have it - an event driven system which uses a global variable to synchronize the execution of our task based on the occurrence of an interrupt. It's still just a bare-metal, OS-baked-into-the-aplication system, but it's introduced a whole bunch of added complexity (and control!) into the system.

The first thing to notice in the source is that the global variable, something_to_do, is used as a synchronization object. When an interrupt occurs from some external event, triggering the My_Interrupt_Source() ISR, program flow in main() is interrupted, the interrupt handler is run, and something_to_do is set to true, letting us know that when we get back to main(), that we should run our Do_something() task.

Another new concept at play here is that of the idle function. In general, when running an event driven system, there are times when the CPU has no application tasks to run. In order to minimize power consumption, CPUs usually contain instructions or registers that can be set up to disable non-essential subsets of the system when there's nothing to do. In general, the sleeping system can be re-activated quickly as a result of an interrupt or other external stimulus, allowing normal processing to resume.

Now, we could just call Do_something() from the interrupt itself - but that's generally not a great solution. In general, the more time we spend inside an interrupt, the more time we spend with at least some interrupts disabled. As a result, we end up with interrupt latency. Now, in this system, with only one interrupt source and only one task this might not be a big deal, but say that Do_something() takes several seconds to complete, and in that time several other interrupts occur from other sources. While executing in our long-running interrupt, no other interrupts can be processed - in many cases, if two interrupts of the same type occur before the first is processed, one of these interrupt events will be lost. This can be utterly disastrous in a real-time system and should be avoided at all costs. As a result, it's generally preferable to use synchronization objects whenever possible to defer processing outside of the ISR.

Another OS concept that is implicitly introduced in this example is that of task priority. When an interrupt occurs, the normal execution of code in main() is preempted: control is swapped over to the ISR (which runs to completion), and then control is given back to main() where it left off. The very fact that interrupts take precedence over what's running shows that main is conceptually a "low-priority" task, and that all ISRs are "high-priority" tasks. In this example, our "high-priority" task is setting a variable to tell our "low-priority" task that it can do something useful. We will investigate the concept of task priority further in the next example.

Preemption is another key principle in embedded systems. This is the notion that whatever the CPU is doing when an interrupt occurs, it should stop, cache its current state (referred to as its context), and allow the high-priority event to be processed. The context of the previous task is then restored its state before the interrupt, and resumes processing. We'll come back to preemption frequently, since the concept comes up frequently in RTOS-based systems.

4.4 Cooperative multi-tasking

Our next example takes the previous example one step further by introducing cooperative multi-tasking:

```
// Bitfield values used to represent three distinct tasks
#define TASK_1_EVENT (0x01)
#define TASK_2_EVENT (0x02)
#define TASK_3_EVENT (0x04)

volatile K_UCHAR event_flags = 0;

// Interrupt sources used to trigger event execution

__interrupt__  My_Interrupt_1(void)
{
    event_flags |= TASK_1_EVENT;
}

__interrupt__ My_Interrupt_2(void)
{
    event_flags |= TASK_2_EVENT;
}

__interrupt__ My_Interrupt_3(void)
{
    event_flags |= TASK_3_EVENT;
}

// Main tasks
int main(void)
{
    while(1)
    {
        while(event_flags)
        {
            if( event_flags & TASK_1_EVENT)
            {
                Do_Task_1();
                event_flags &= ~TASK_1_EVENT;
            } else if( event_flags & TASK_2_EVENT) {
                Do_Task_2();
                event_flags &= ~TASK_2_EVENT;
            } else if( event_flags & TASK_3_EVENT) {
                Do_Task_3();
                event_flags &= ~TASK_3_EVENT;
            }
        }
        Idle();
    }
}
```

This system is very similar to what we had before - however the differences are worth discussing. First, we have stimulus from multiple interrupt sources: each ISR is responsible for setting a single bit in our global event flag, which is then used to control execution of individual tasks from within main().

Next, we can see that tasks are explicitly given priorities inside the main loop based on the logic of the if/else if structure. As long as there is something set in the event flag, we will always try to execute Task1 first, and only when Task1 isn't set will we attempt to execute Task2, and then Task 3. This added logic provides the notion of priority. However, because each of these tasks exist within the same context (they're just different functions called from our main control loop), we don't have the same notion of preemption that we have when dealing with interrupts.

That means that even through we may be running Task2 and an event flag for Task1 is set by an interrupt, the CPU still has to finish processing Task2 to completion before Task1 can be run. And that's why this kind of scheduling is referred to ascooperative multitasking: we can have as many tasks as we want, but unless they cooperate by means of returning back to main, the system can end up with high-priority tasks getting starved for CPU time by lower-priority, long-running tasks.

This is one of the more popular Os-baked-into-the-application approaches, and is widely used in a variety of real-time embedded systems.

4.5 Hybrid cooperative/preemptive multi-tasking

The final variation on the superloop design utilizes software-triggered interrupts to simulate a hybrid cooperative/preemptive multitasking system. Consider the example code below.

```
// Bitfields used to represent high-priority tasks.  Tasks in this group
// can preempt tasks in the group below - but not eachother.
#define HP_TASK_1        (0x01)
#define HP_TASK_2        (0x02)

volatile K_UCHAR hp_tasks = 0;

// Bitfields used to represent low-priority tasks.
#define LP_TASK_1        (0x01)
#define LP_TASK_2        (0x02)

volatile K_UCHAR lp_tasks = 0;

// Interrupt sources, used to trigger both high and low priority tasks.
__interrupt__ System_Interrupt_1(void)
{
    // Set any of the other tasks from here...
    hp_tasks |= HP_TASK_1;
    // Trigger the SWI that calls the High_Priority_Tasks interrupt handler
    SWI();
}

__interrupt__ System_Interrupt_n...(void)
{
    // Set any of the other tasks from here...
}


// Interrupt handler that is used to implement the high-priority event context
__interrupt__ High_Priority_Tasks(void)
{
    // Enabled every interrupt except this one
    Disable_My_Interrupt();
    Enable_Interrupts();
    while( hp_tasks)
    {
        if( hp_tasks & HP_TASK_1)
        {
            HP_Task1();
            hp_tasks &= ~HP_TASK_1;
        }
        else if (hp_tasks & HP_TASK_2)
        {
            HP_Task2();
            hp_tasks &= ~HP_TASK_2;
        }
    }
    Restore_Interrupts();
    Enable_My_Interrupt();
}
```

```
// Main loop, used to implement the low-priority events
int main(void)
{
    // Set the function to run when a SWI is triggered
    Set_SWI(High_Priority_Tasks);

    // Run our super-loop
    while(1)
    {
        while (lp_tasks)
        {
            if (lp_tasks & LP_TASK_1)
            {
                LP_Task1();
                lp_tasks &= ~LP_TASK_1;
            }
            else if (lp_tasks & LP_TASK_2)
            {
                LP_Task2();
                lp_tasks &= ~LP_TASK_2;
            }
        }
        Idle();
    }
}
```

In this example, High_Priority_Tasks() can be triggered at any time as a result of a software interrupt (SWI),. When a high-priority event is set, the code that sets the event calls the SWI as well, which instantly preempts whatever is happening in main, switching to the high-priority interrupt handler. If the CPU is executing in an interrupt handler already, the current ISR completes, at which point control is given to the high priority interrupt handler.

Once inside the HP ISR, all interrupts (except the software interrupt) are re-enabled, which allows this interrupt to be preempted by other interrupt sources, which is called interrupt nesting. As a result, we end up with two distinct execution contexts (main and HighPriorityTasks()), in which all tasks in the high-priority group are guaranteed to preempt main() tasks, and will run to completion before returning control back to tasks in main(). This is a very basic preemptive multitasking scenario, approximating a "real" RTOS system with two threads of different priorities.

4.6 Problems with superloops

As mentioned earlier, a lot of real-world systems are implemented using a superloop design; and while they are simple to understand due to the limited and obvious control logic involved, they are not without their problems.

Hidden Costs

It's difficult to calculate the overhead of the superloop and the code required to implement workarounds for blocking calls, scheduling, and preemption. There's a cost in both the logic used to implement workarounds (usually involving state machines), as well as a cost to maintainability that comes with breaking up into chunks based on execution time instead of logical operations. In moderate firmware systems, this size cost can exceed the overhead of a reasonably well-featured RTOS, and the deficit in maintainability is something that is measurable in terms of lost productivity through debugging and profiling.

Tightly-coupled code

Because the control logic is integrated so closely with the application logic, a lot of care must be taken not to compromise the separation between application and system code. The timing loops, state machines, and architecture-specific control mechanisms used to avoid (or simulate) preemption can all contribute to the problem. As a result, a lot of superloop code ends up being difficult to port without effectively simulating or replicating the underlying system for which the application was written. Abstraction layers can mitigate the risks, but a lot of care should be taken to fully decouple the application code from the system code.

No blocking calls

In a super-loop environment, there's no such thing as a blocking call or blocking objects. Tasks cannot stop mid-execution for event-driven I/O from other contexts - they must always run to completion. If busy-waiting and polling are used as a substitute, it increases latency and wastes cycles. As a result, extra code complexity is often times necessary to work-around this lack of blocking objects, often times through implementing additional state machines. In a large enough system, the added overhead in code size and cycles can add up.

Difficult to guarantee responsiveness

Without multiple levels of priority, it may be difficult to guarantee a certain degree of real-time responsiveness without added profiling and tweaking. The latency of a given task in a priority-based cooperative multitasking system is the length of the longest task. Care must be taken to break tasks up into appropriate sized chunks in order to ensure that higher-priority tasks can run in a timely fashion - a manual process that must be repeated as new tasks are added in the system. Once again, this adds extra complexity that makes code larger, more difficult to understand and maintain due to the artificial subdivision of tasks into time-based components.

Limited preemption capability

As shown in the example code, the way to gain preemption in a superloop is through the use of nested interrupts. While this isn't unwiedly for two levels of priority, adding more levels beyond this is becomes complicated. In this case, it becomes necessary to track interrupt nesting manually, and separate sets of tasks that can run within given priority loops - and deadlock becomes more difficult to avoid.

Chapter 5

Mark3 Overview

5.1 Intro

The following section details the overall design of Mark3, the goals I've set out to achieve, the features that I've intended to provide, as well as an introduction to the programming concepts used to make it happen.

5.2 Features

Mark3 is a fully-featured real-time kernel, and is feature-competitive with other open-source and commercial RTOS's in the embedded arena.

The key features of this RTOS are:

- Flexible Scheduler
  - Unlimited number of threads with 8 priority levels
  - Unlimited threads per priority level
  - Round-robin scheduling for threads at each priority level
  - Time quantum scheduling for each thread in a given priority level
- Configurable stacks for each Thread
- Resource protection:
  - Integrated mutual-exclusion semaphores (Mutex)
  - Priority-inheritance on Mutex objects to prevent priority inversion
- Synchronization Objects
  - Binary and counting Semaphore to coordinate thread execution
  - Event flags with 16-bit bitfields for complex thread synchronization
- Efficient Timers
  - The RTOS is tickless, the OS only wakes up when a timer expires, not at a regular interval
  - One-shot and periodic timers with event callbacks
  - Timers are high-precision and long-counting (about 68000 seconds when used with a 16us resolution timer)
- Driver API
  - A hardware abstraction layer is provided to simplify driver development

- Robust Interprocess Communications
  - Threadsafe global Message pool and configurable message queues
- Support for kernel-aware simulation
  - Provides advanced test and verification functionality, allowing for easy integration into continuous-integration systems
  - Provide accurate engineering data on key metrics like stack usage and realtime performance, with easy-to-use APIs and little overhead

5.3 Design Goals

Lightweight

Mark3 can be configured to have an extremely low static memory footprint. Each thread is defined with its own stack, and each thread structure can be configured to take as little as 26 bytes of RAM. The complete Mark3 kernel with all features, setup code, a serial driver, and the Mark3 protocol libraries comes in at under 9K of code space and 1K of RAM on atmel AVR.

Modular

Each system feature can be enabled or disabled by modifying the kernel configuration header file. Include what you want, and ignore the rest to save code space and RAM.

Easily Portable

Mark3 should be portable to a variety of 8, 16 and 32 bit architectures without MMUs. Porting the OS to a new architecture is relatively straightforward, requiring only device-specific implementations for the lowest-level operations such as context switching and timer setup.

Easy To Use

Mark3 is small by design - which gives it the advantage that it's also easy to develop for. This manual, the code itself, and the Doxygen documentation in the code provide ample documentation to get you up to speed quickly. Because you get to see the source, there's nothing left to assumption.

Simple to Understand

Not only is the Mark3 API rigorously documented (hey - that's what this book is for!), but the architecture and naming conventions are intuitive - it's easy to figure out where code lives, and how it works. Individual modules are small due to the "one feature per file" rule used in development. This makes Mark3 an ideal platform for learning about aspects of RTOS design.

Chapter 6

Getting Started

6.1 Kernel Setup

This section details the process of defining threads, initializing the kernel, and adding threads to the scheduler.

If you're at all familiar with real-time operating systems, then these setup and initialization steps should be familiar. I've tried very hard to ensure that as much of the heavy lifting is hidden from the user, so that only the bare minimum of calls are required to get things started.

The examples presented in this chapter are real, working examples taken from the ATmega328p port.

First, you'll need to create the necessary data structures and functions for the threads:

1. Create a Thread object for all of the "root" or "initial" tasks.
2. Allocate stacks for each of the Threads
3. Define an entry-point function for each Thread

This is shown in the example code below:

```
//-----------------------------------------------------------------------------
#include "thread.h"
#include "kernel.h"

//1) Create a thread object for all of the "root" or "initial" tasks
static Thread AppThread;
static Thread IdleThread;

//2) Allocate stacks for each thread
#define STACK_SIZE_APP      (192)
#define STACK_SIZE_IDLE     (128)

static K_UCHAR aucAppStack[STACK_SIZE_APP];
static K_UCHAR aucIdleStack[STACK_SIZE_IDLE];

//3) Define entry point functions for each thread
void AppThread(void);
void IdleThread(void);
```

Next, we'll need to add the required kernel initialization code to main. This consists of running the Kernel's init routine, initializing all of the threads we defined, adding the threads to the scheduler, and finally calling Kernel::-Start(), which transfers control of the system to the RTOS.

These steps are illustrated in the following example.

```
int main(void)
{
    //1) Initialize the kernel prior to use
    Kernel::Init();             // MUST be before other kernel ops

    //2) Initialize all of the threads we've defined
```

```
    AppThread.Init( aucAppStack,        // Pointer to the stack
                    STACK_SIZE_APP,     // Size of the stack
                    1,          // Thread priority
                    (void*)AppEntry,    // Entry function
                    NULL );             // Entry function argument

    IdleThread.Init( aucIdleStack,      // Pointer to the stack
                     STACK_SIZE_IDLE,   // Size of the stack
                     0,         // Thread priority
                     (void*)IdleEntry,  // Entry function
                     NULL );        // Entry function argument

    //3) Add the threads to the scheduler
    AppThread.Start();          // Actively schedule the threads
    IdleThread.Start();

    //4) Give control of the system to the kernel
    Kernel::Start();            // Start the kernel!
}
```

Not much to it, is there? There are a few noteworthy points in this code, though.

In order for the kernel to work properly, a system must always contain an idle thread; that is, a thread at priority level 0 that never blocks. This thread is responsible for performing any of the low-level power management on the CPU in order to maximize battery life in an embedded device. The idle thread must also never block, and it must never exit. Either of these operations will cause undefined behavior in the system.

The App thread is at a priority level greater-than 0. This ensures that as long as the App thread has something useful to do, it will be given control of the CPU. In this case, if the app thread blocks, control will be given back to the Idle thread, which will put the CPU into a power-saving mode until an interrupt occurs.

Stack sizes must be large enough to accommodate not only the requirements of the threads, but also the requirements of interrupts - up to the maximum interrupt-nesting level used. Stack overflows are super-easy to run into in an embedded system; if you encounter strange and unexplained behavior in your code, chances are good that one of your threads is blowing its stack.

6.2 Threads

Mark3 Threads act as independent tasks in the system. While they share the same address-space, global data, device-drivers, and system peripherals, each thread has its own set of CPU registers and stack, collectively known as the thread's **context**. The context is what allows the RTOS kernel to rapidly switch between threads at a high rate, giving the illusion that multiple things are happening in a system, when really, only one thread is executing at a time.

6.2.1 Thread Setup

Each instance of the Thread class represents a thread, its stack, its CPU context, and all of the state and metadata maintained by the kernel. Before a Thread will be scheduled to run, it must first be initialized with the necessary configuration data.

The Init function gives the user the opportunity to set the stack, stack size, thread priority, entry-point function, entry-function argument, and round-robin time quantum:

Thread stacks are pointers to blobs of memory (usually K_CHAR arrays) carved out of the system's address space. Each thread must have a stack defined that's large enough to handle not only the requirements of local variables in the thread's code path, but also the maximum depth of the ISR stack.

Priorities should be chosen carefully such that the shortest tasks with the most strict determinism requirements are executed first - and are thus located in the highest priorities. Tasks that take the longest to execute (and require the least degree of responsiveness) must occupy the lower thread priorities. The idle thread must be the only thread occupying the lowest priority level.

The thread quantum only aplies when there are multiple threads in the ready queue at the same priority level. This interval is used to kick-off a timer that will cycle execution between the threads in the priority list so that they each get a fair chance to execute.

The entry function is the function that the kernel calls first when the thread instance is first started. Entry functions have at most one argument - a pointer to a data-object specified by the user during initialization.

An example thread initailization is shown below:

```
Thread clMyThread;
K_UCHAR aucStack[192];

void AppEntry(void)
{
    while(1)
    {
        // Do something!
    }
}

...
{
    clMyThread.Init(aucStack,    // Pointer to the stack to use by this thread
                    192,             // Size of the stack in bytes
                    1,       // Thread priority (0 = idle, 7 = max)
                    (void*)AppEntry, // Function where the thread starts executing
                    NULL );          // Argument passed into the entry function

}
```

Once a thread has been initialized, it can be added to the scheduler by calling:

```
clMyThread.Start();
```

The thread will be placed into the Scheduler's queue at the designated priority, where it will wait its turn for execution.

6.2.2 Entry Functions

Mark3 Threads should not run-to-completion - they should execute as infinite loops that perform a series of tasks, appropriately partitioned to provide the responsiveness characteristics desired in the system.

The most basic Thread loop is shown below:

```
void Thread( void *param )
{
    while(1)
    {
        // Do Something
    }
}
```

Threads can interact with eachother in the system by means of synchronization objects (Semaphore), mutual-exclusion objects (Mutex), Inter-process messaging (MessageQueue), and timers (Timer).

Threads can suspend their own execution for a predetermined period of time by using the static Thread::Sleep() method. Calling this will block the Thread's executin until the amount of time specified has ellapsed. Upon expiry, the thread will be placed back into the ready queue for its priority level, where it awaits its next turn to run.

6.3 Timers

Timer objects are used to trigger callback events periodic or on a one-shot (alarm) basis.

While extremely simple to use, they provide one of the most powerful execution contexts in the system. The timer callbacks execute from within the timer callback ISR in an interrupt-enabled context. As such, timer callbacks are considered higher-priority than any thread in the system, but lower priority than other interrupts. Care must be taken to ensure that timer callbacks execute as quickly as possible to minimize the impact of processing on the throughput of tasks in the system. Wherever possible, heavy-lifting should be deferred to the threads by way of semaphores or messages.

Below is an example showing how to start a periodic system timer which will trigger every second:

```
{
    Timer clTimer;
    clTimer.Init();

    clTimer.Start( 1000,
                   1,
                   MyCallback,
                   (void*)&my_data );

    ... // Keep doing work in the thread
}

// Callback function, executed from the timer-expiry context.
void MyCallBack( Thread *pclOwner_, void *pvData_ )
{
    LED.Flash(); // Flash an LED.
}
```

6.4 Semaphores

Semaphores are used to synchronized execution of threads based on the availability (and quantity) of application-specific resources in the system. They are extremely useful for solving producer-consumer problems, and are the method-of-choice for creating efficient, low latency systems, where ISRs post semaphores that are handled from within the context of individual threads. (Yes, Semaphores can be posted - but not pended - from the interrupt context).

The following is an example of the producer-consumer usage of a binary semaphore:

```
Semaphore clSemaphore; // Declare a semaphore shared between a producer and a consumer thread.

void Producer()
{
    clSemaphore.Init(0, 1);
    while(1)
    {
        // Do some work, create something to be consumed

        // Post a semaphore, allowing another thread to consume the data
        clSemaphore.Post();
    }
}

void Consumer()
{
    // Assumes semaphore initialized before use...
    While(1)
    {
        // Wait for new data from the producer thread
        clSemaphore.Pend();

        // Consume the data!
    }
}
```

And an example of using semaphores from the ISR context to perform event- driven processing.

```
Semaphore clSemaphore;

__interrupt__ MyISR()
{
    clSemaphore.Post(); // Post the interrupt.  Lightweight when uncontested.
}

void MyThread()
{
    clSemaphore.Init(0, 1); // Ensure this is initialized before the MyISR interrupt is enabled.
    while(1)
    {
        // Wait until we get notification from the interrupt
        clSemaphore.Pend();

        // Interrupt has fired, do the necessary work in this thread's context
        HeavyLifting();
    }
}
```

6.5 Mutexes

Mutexes (Mutual exclusion objects) are provided as a means of creating "protected sections" around a particular resource, allowing for access of these objects to be serialized. Only one thread can hold the mutex at a time - other threads have to wait until the region is released by the owner thread before they can take their turn operating on the protected resource. Note that mutexes can only be owned by threads - they are not available to other contexts (i.e. interrupts). Calling the mutex APIs from an interrupt will cause catastrophic system failures.

Note that these objects are also not recursive- that is, the owner thread can not attempt to claim a mutex more than once.

Prioritiy inheritence is provided with these objects as a means to avoid prioritiy inversions. Whenever a thread at a priority than the mutex owner blocks on a mutex, the priority of the current thread is boosted to the highest-priority waiter to ensure that other tasks at intermediate priorities cannot artificically prevent progress from being made.

Mutex objects are very easy to use, as there are only three operations supported: Initialize, Claim and Release. An example is shown below.

```
Mutex clMutex;  // Create a mutex globally.

void Init()
{
    // Initialize the mutex before use.
    clMutex.Init();
}

// Some function called from a thread
void Thread1Function()
{
    clMutex.Claim();

    // Once the mutex is owned, no other thread can
    // enter a block protect by the same mutex

    my_protected_resource.do_something();
    my_protected_resource.do_something_else();

    clMutex.Release();
}

// Some function called from another thread
void Thread2Function()
{
    clMutex.Claim();

    // Once the mutex is owned, no other thread can
    // enter a block protect by the same mutex

    my_protected_resource.do_something();
    my_protected_resource.do_different_things();

    clMutex.Release();
}
```

6.6 Event Flags

Event Flags are another synchronization object, conceptually similar to a semaphore.

Unlike a semaphore, however, the condition on which threads are unblocked is determined by a more complex set of rules. Each Event Flag object contains a 16-bit field, and threads block, waiting for combinations of bits within this field to become set.

A thread can wait on any pattern of bits from this field to be set, and any number of threads can wait on any number of different patterns. Threads can wait on a single bit, multiple bits, or bits from within a subset of bits within the field.

As a result, setting a single value in the flag can result in any number of threads becoming unblocked simultaneously. This mechanism is extremely powerful, allowing for all sorts of complex, yet efficient, thread synchronization schemes that can be created using a single shared object.

Note that Event Flags can be set from interrupts, but you cannot wait on an event flag from within an interrupt.

Examples demonstrating the use of event flags are shown below.

```
// Simple example showing a thread blocking on a multiple bits in the
// fields within an event flag.

EventFlag clEventFlag;

int main()
{
    ...
    clEventFlag.Init(); // Initialize event flag prior to use
    ...
}

void MyInterrupt()
{
    // Some interrupt corresponds to event 0x0020
    clEventFlag.Set(0x0020);
}

void MyThreadFunc()
{
    ...
    while(1)
    {
        ...
        K_USHORT usWakeCondition;

        // Allow this thread to block on multiple flags
        usWakeCondition = clEventFlag.Wait(0x00FF, EVENT_FLAG_ANY);

        // Clear the event condition that caused the thread to wake (in this case,
        // usWakeCondtion will equal 0x20 when triggered from the interrupt above)
        clEventFlag.Clear(usWakeCondition);

        // <do something>
    }
}
```

6.7 Messages

Sending messages between threads is the key means of synchronizing access to data, and the primary mechanism to perform asynchronous data processing operations.

Sending a message consists of the following operations:

- Obtain a Message object from the global message pool
- Set the message data and event fields
- Send the message to the destination message queue

While receiving a message consists of the following steps:

- Wait for a messages in the destination message queue
- Process the message data
- Return the message back to the global message pool

These operations, and the various data objects involved are discussed in more detail in the following section.

6.7.1 Message Objects

Message objects are used to communicate arbitrary data between threads in a safe and synchronous way.

The message object consists of an event code field and a data field. The event code is used to provide context to the message object, while the data field (essentially a void * data pointer) is used to provide a payload of data corresponding to the particular event.

Access to these fields is marshalled by accessors - the transmitting thread uses the SetData() and SetCode() methods to seed the data, while the receiving thread uses the GetData() and GetCode() methods to retrieve it.

By providing the data as a void data pointer instead of a fixed-size message, we achieve an unprecedented measure of simplicity and flexibility. Data can be either statically or dynamically allocated, and sized appropriately for the event without having to format and reformat data by both sending and receiving threads. The choices here are left to the user - and the kernel doesn't get in the way of efficiency.

It is worth noting that you can send messages to message queues from within ISR context. This helps maintain consistency, since the same APIs can be used to provide event-driven programming facilities throughout the whole of the OS.

6.7.2 Global Message Pool

To maintain efficiency in the messaging system (and to prevent over-allocation of data), a global pool of message objects is provided. The size of this message pool is specified in the implementation, and can be adjusted depending on the requirements of the target application as a compile-time option.

Allocating a message from the message pool is as simple as calling the GlobalMessagePool::Pop() Method.

Messages are returned back to the GlobalMessagePool::Push() method once the message contents are no longer required.

One must be careful to ensure that discarded messages always are returned to the pool, otherwise a resource leak can occur, which may cripple the operating system's ability to pass data between threads.

6.7.3 Message Queues

Message objects specify data with context, but do not specify where the messages will be sent. For this purpose we have a MessageQueue object. Sending an object to a message queue involves calling the MessageQueue::Send() method, passing in a pointer to the Message object as an argument.

When a message is sent to the queue, the first thread blocked on the queue (as a result of calling the MessageQueue Receive() method) will wake up, with a pointer to the Message object returned.

It's worth noting that multiple threads can block on the same message queue, providing a means for multiple threads to share work in parallel.

6.7.4 Messaging Example

```
// Message queue object shared between threads
MessageQueue clMsgQ;

// Function that initializes the shared message queue
void MsgQInit()
{
    clMsgQ.Init();
}

// Function called by one thread to send message data to
// another
void TxMessage()
{
    // Get a message, initialize its data
    Message *pclMesg = GlobalMessagePool::Pop();

    pclMesg->SetCode(0xAB);
    pclMesg->SetData((void*)some_data);

    // Send the data to the message queue
    clMsgQ.Send(pclMesg);
}

// Function called in the other thread to block until
// a message is received in the message queue.
void RxMessage()
{
    Message *pclMesg;
```

```
    // Block until we have a message in the queue
    pclMesg = clMsgQ.Receive();

    // Do something with the data once the message is received
    pclMesg->GetCode();

    // Free the message once we're done with it.
    GlobalMessagePool::Push(pclMesg);
}
```

6.8 Sleep

There are instances where it may be necessary for a thread to poll a resource, or wait a specific amount of time before proceeding to operate on a peripheral or volatile piece of data.

While the Timer object is generally a better choice for performing time-sensitive operations (and certainly a better choice for periodic operations), the Thread::Sleep() method provides a convenient (and efficient) mechanism that allows for a thread to suspend its execution for a specified interval.

Note that when a thread is sleeping it is blocked, during which other threads can operate, or the system can enter its idle state.

```
int GetPeripheralData();
{
    int value;
    // The hardware manual for a peripheral specifies that
    // the "foo()" method will result in data being generated
    // that can be captured using the "bar()" method.
    // However, the value only becomes valid after 10ms

    peripheral.foo();
    Thread::Sleep(10);  // Wait 10ms for data to become valid
    value = peripheral.bar();
    return value;
}
```

6.9 Round-Robin Quantum

Threads at the same thread priority are scheduled using a round-robin scheme. Each thread is given a timeslice (which can be configured) of which it shares time amongst ready threads in the group. Once a thread's timeslice has expired, the next thread in the priority group is chosen to run until its quantum has expired - the cycle continues over and over so long as each thread has work to be done.

By default, the round-robin interval is set at 4ms.

This value can be overridden by calling the thread's SetQuantum() with a new interval specified in milliseconds.

Chapter 7

Build System

Mark3 is distributed with a recursive makefile build system, allowing the entire source tree to be built into a series of libraries with simple make commands.

The way the scripts work, every directory with a valid makefile is scanned, as well as all of its subdirectories. The build then generates binary components for all of the components it finds -libraries and executables. All libraries that are generated can then be imported into an application using the linker without having to copy-and-paste files on a module-by-module basis. Applications built during this process can then be loaded onto a device directly, without requiring a GUI-based IDE. As a result, Mark3 integrates well with 3rd party tools for continuous-integration and automated testing.

This modular framework allows for large volumes of libraries and binaries to be built at once - the default build script leverages this to build all of the examples and unit tests at once, linking against the pre-built kernel, services, and drivers. Whatever can be built as a library is built as a library, promoting reuse throughout the platform, and enabling Mark3 to be used as a platform, with an ecosystem of libraries, services, drivers and applications.

7.1 Source Layout

One key aspect of Mark3 is that system features are organized into their own separate modules. These modules are further grouped together into folders based on the type of features represented:

```
Root         Base folder, contains recursive makefiles for build system
    arduino      Arduino-specific headers and API documentation files
    bootloader   Mark3 Bootloader code for AVR microcontrollers
    build        Makefiles and device-configuraton data for various platforms
    docs         Documentation (including this)
    drivers      Device driver code for various supported devices
    example      Example applications
    fonts        Bitmap fonts converted from TTF, used by Mark3 graphics library
    kernel       Basic Mark3 Components (the focus of this manual)
        cpu      CPU-specific porting code
    scripts      Scripts used to simplify build, documentation, and profiling
    services     Utility code and services, extended system features
    stage        Staging directory, where the build system places artifacts
    tests        Unit tests, written as C/C++ applications
    util         .net-based utils: font conversion, terminal, programmer, and configuration
```

7.2 Building the kernel

The base.mak file determines how the kernel, drivers, and libraries are built, for what targets, and with what options. Most of these options can be copied directly from the options found in your IDE managed projects. Below is an overview of the main variables used to configure the build.

```
STAGE       - Location in the filesystem where the build output is stored
```

```
ROOT_DIR    - The location of the root source tree
ARCH        - The CPU architecture to build against
VARIANT     - The variant of the above CPU to target
TOOLCHAIN   - Which toolchain to build with (dependent on ARCH and VARIANT)
```

Build.mak contains the logic which is used to perform the recursive make in all directories. Unless you really know what you're doing, it's best to leave this as-is.

You must make sure that all required paths are set in your system environment variables so that they are accessible through from the command-line.

Once configured, you can build the source tree using the various make targets:

- make headers
  - copy all headers in each module's /public subdirectory to the location specified by STAGE environment variable's ./inc subdirectory.
- make library
  - regenerate all objects copy marked as libraries (i.e. the kernel + drivers). Resulting binaries are copied into STAGE's ./lib subdirectory.
- make binary
  - build all executable projects in the root directory structure. In the default distribution, this includes the basic set of demos.

These steps are chained together automatically as part of the build.sh script found under the /scripts subdirectory. Running ./scripts/build.sh from the root of the embedded source directory will result in all headers being exported, libraries built, and applications built. This script will also default to building for atmega328p using GCC if none of the required environment variables have previously been configured.

To add new components to the recursive build system, simply add your code into a new folder beneath the root install location.

Source files, the module makefile and private header files go directly in the new folder, while public headers are placed in a ./public subdirectory. Create a ./obj directory to hold the output from the builds.

The contents of the module makefile looks something like this:

```
# Include common prelude make file
include $(ROOT_DIR)base.mak

# If we're building a library, set IS_LIB and LIBNAME
# If we're building an app, set IS_APP and APPNAME
IS_LIB=1
LIBNAME=mylib

#this is the list of the source modules required to build the kernel
CPP_SOURCE = mylib.cpp \
             someotherfile.cpp

# Similarly, C-language source would be under the C_SOURCE variable.

# Include the rest of the script that is actually used for building the
# outputs
include $(ROOT_DIR)build.mak
```

Once you've placed your code files in the right place, and configured the makefile appropriately, a fresh call to make headers, make library, then make binary will guarantee that your code is built.

Now, you can still copy-and-paste the required kernel, port, and drivers, directly into your application avoiding the whole process of using make from the command line. To do this, run "make source" from the root directory in svn, and copy the contents of /stage/src into your project. This should contain the source to the kernel, all drivers, and all services that are in the tree - along with the necessary header files.

7.3 Building on Windows

Building Mark3 on Windows is the same as on Linux, but there are a few prerequisites that need to be taken into consideration before the build scripts and makefiles will work as expected.

Step 1 - Install Latest Atmel Studio IDE

Atmel Studio contains the AVR8 GCC toolchain, which contains the necessary compilers, assemblers, and platform support required to turn the source modules into libraries and executables.

To get Atmel Studio, go to the Atmel website (http://www.atmel.com) and register to download the latest version. This is a free download (and rather large). The included IDE (if you choose to use it) is very slick, as it's based on Visual Studio, and contains a wonderful cycle-accurate simulator for AVR devices. In fact, the simulator is so good that most of the kernel and its drivers were developed using this tool.

Once you have downloaded and installed Atmel Studio, you will need to add the location of the AVR toolcahin to the PATH environment variable.

To do this, go to Control Panel -> System and Security -> System -> Advanced System Settings, and edit the PATH variable. Append the location of the toolchain bin folder to the end of the variable.

On Windows 7 x64, it should look something like this:

C: Files (x86) Toolchain GCC\Native\3.4.2.1002-gnu-toolchain

Step 2 - Install MinGW and MinSys

MinGW (and MinSys in particular) provide a unix-like environment that runs under windows. Some of the utilities provided include a version of the bash shell, and GNU standard make - both which are required by the Mark3 recursive build system.

The MinGW installer can be downloaded from its project page on SourceForge. When installing, be sure to select the "MinSys" component.

Once installed, add the MinSys binary path to the PATH environment variable, in a similar fashion as with Atmel Studio in Step 1.

Step 3 - Setup Include Paths in Platform Makefile

The AVR header file path must be added to the "platform.mak" makefile for each AVR Target you are attempting to build for. These files can be located under /embedded/build/avr/atmegaXXX/. The path to the includes directory should be added to the end of the CFLAGS and CPPFLAGS variables, as shown in the following:

```
TEST_INC="/c/Program Files (x86)/Atmel/Atmel Toolchain/AVR8
       GCC/Native/3.4.2.1002/avr8-gnu-toolchain/include"
CFLAGS += -I$(TEST_INC)
CPPFLAGS += -I$(TEST_INC)
```

Step 4 - Build Mark3 using Bash

Launch a terminal to your Mark3 base directory, and cd into the "embedded" folder. You should now be able to build Mark3 by running "bash ./build.sh" from the command-line.

Alternately, you can run bash itself, building Mark3 by running ./build.sh or the various make targets using the same synatx as documented previously.

Note - building on Windows is *slow*. This has a lot to do with how "make" performs under windows. There are faster substitutes for make (such as cs-make) that are exponentially quicker, and approach the performance of make on Linux. Other mechanisms, such as running make with multiple concurrent jobs (i.e. "make -j4") also helps significantly, especially on systems with multicore CPUs.

Chapter 8

License

8.1 License

Chapter 9

Profiling Results

The following profiling results were obtained using an ATMega328p @ 16MHz.

The test cases are designed to make use of the kernel profiler, which accurately measures the performance of the fundamental system APIs, in order to provide information for user comparison, as well as to ensure that regressions are not being introduced into the system.

9.1 Date Performed

Thu Mar 5 21:04:16 EST 2015

9.2 Compiler Information

The kernel and test code used in these results were built using the following compiler: Using built-in specs. COLLECT_GCC=avr-gcc COLLECT_LTO_WRAPPER=/usr/lib/gcc/avr/4.8.2/lto-wrapper Target: avr Configured with: ../src/configure -v –enable-languages=c,c++ –prefix=/usr/lib –infodir=/usr/share/info –mandir=/usr/share/man –bindir=/usr/bin –libexecdir=/usr/lib –libdir=/usr/lib –enable-shared –with-system-zlib –enable-long-long –enable-nls –without-included-gettext –disable-libssp –build=x86_64-linux-gnu –host=x86_64-linux-gnu –target=avr Thread model: single gcc version 4.8.2 (GCC)

9.3 Profiling Results

- Semaphore Initialization: 40 cycles (averaged over 169 iterations)
- Semaphore Post (uncontested): 111 cycles (averaged over 169 iterations)
- Semaphore Pend (uncontested): 78 cycles (averaged over 169 iterations)
- Semaphore Flyback Time (Contested Pend): 1575 cycles (averaged over 169 iterations)
- Mutex Init: 223 cycles (averaged over 169 iterations)
- Mutex Claim: 223 cycles (averaged over 169 iterations)
- Mutex Release: 119 cycles (averaged over 169 iterations)
- Thread Initialize: 8280 cycles (averaged over 169 iterations)
- Thread Start: 775 cycles (averaged over 169 iterations)
- Context Switch: 191 cycles (averaged over 168 iterations)
- Thread Schedule: 95 cycles (averaged over 168 iterations)

Chapter 10

Code Size Profiling

The following report details the size of each module compiled into the kernel.

The size of each component is dependent on the flags specified in mark3cfg.h at compile time. Note that these sizes represent the maximum size of each module before dead code elimination and any additional link-time optimization, and represent the maximum possible size that any module can take.

The results below are for profiling on Atmel AVR atmega328p-based targets using gcc. Results are not necessarily indicative of relative or absolute performance on other platforms or toolchains.

10.1 Information

Subversion Repository Information:

- Repository Root: svn+ssh://m0slevin.code.sf.net/p/mark3/source
- Revision: 188
- URL: svn+ssh://m0slevin.code.sf.net/p/mark3/source/branch/release/R1/embedded Relative URL: ^/branch/release/-R1/embedded

Date Profiled: Thu Mar 5 21:04:20 EST 2015

10.2 Compiler Version

avr-gcc (GCC) 4.8.2 Copyright (C) 2013 Free Software Foundation, Inc. This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

10.3 Profiling Results

Mark3 Module Size Report:

- Synchronization Objects - Base Class..... : 84 Bytes
- Device Driver Framework (including /dev/null)... : 226 Bytes
- Synchronization Object - Event Flag..... : 770 Bytes
- Fundamental Kernel Linked-List Classes..... : 496 Bytes

- Message-based IPC..... : 426 Bytes
- Mutex (Synchronization Object)..... : 658 Bytes
- Performance-profiling timers..... : 546 Bytes
- Round-Robin Scheduling Support..... : 252 Bytes
- Thread Scheduling..... : 475 Bytes
- Semaphore (Synchronization Object)..... : 544 Bytes
- Thread Implementation..... : 1433 Bytes
- Fundamental Kernel Thread-list Data Structures.. : 212 Bytes
- Mark3 Kernel Base Class..... : 80 Bytes
- Software Timer Implementation..... : 1015 Bytes
- Runtime Kernel Trace Implementation..... : 0 Bytes
- Circular Logging Buffer Base Class..... : 0 Bytes
- Atmel AVR - Kernel Aware Simulation Support..... : 287 Bytes
- Atmel AVR - Basic Threading Support..... : 528 Bytes
- Atmel AVR - Kernel Interrupt Implemenation..... : 56 Bytes
- Atmel AVR - Kernel Timer Implementation..... : 322 Bytes
- Atmel AVR - Profiling Timer Implementation..... : 256 Bytes

Mark3 Kernel Size Summary:

- Kernel : 2780 Bytes
- Synchronization Objects : 2398 Bytes
- Port : 1449 Bytes
- Features : 2039 Bytes
- Total Size : 8666 Bytes

Chapter 11

Hierarchical Index

11.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 12

Class Index

12.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 13

File Index

13.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 14

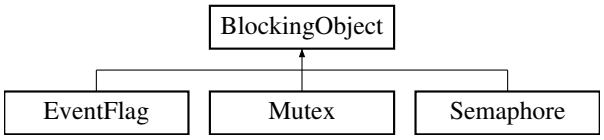
Class Documentation

14.1 BlockingObject Class Reference

Class implementing thread-blocking primatives.

`#include <blocking.h>`

Inheritance diagram for BlockingObject:



Protected Member Functions

- void Block (Thread ∗pclThread_)
- void UnBlock (Thread ∗pclThread_)

Protected Attributes

- ThreadList m_clBlockList

  ThreadList which is used to hold the list of threads blocked on a given object.

14.1.1 Detailed Description

Class implementing thread-blocking primatives.

Used for implementing things like semaphores, mutexes, message queues, or anything else that could cause a thread to suspend execution on some external stimulus.

Definition at line 65 of file blocking.h.

14.1.2 Member Function Documentation

14.1.2.1 void BlockingObject::Block (Thread ∗ *pclThread_*) [protected]

Parameters

*pclThread_*	Pointer to the thread object that will be blocked.

Blocks a thread on this object. This is the fundamental operation performed by any sort of blocking operation in the operating system. All semaphores/mutexes/sleeping/messaging/etc ends up going through the blocking code at some point as part of the code that manages a transition from an "active" or "waiting" thread to a "blocked" thread.

The steps involved in blocking a thread (which are performed in the function itself) are as follows;

1) Remove the specified thread from the current owner's list (which is likely one of the scheduler's thread lists) 2) Add the thread to this object's thread list 3) Setting the thread's "current thread-list" point to reference this object's threadlist.

Definition at line 36 of file blocking.cpp.

14.1.2.2 void BlockingObject::UnBlock (Thread * *pclThread_*) [protected]

Parameters

*pclThread_*	Pointer to the thread to unblock.

Unblock a thread that is already blocked on this object, returning it to the "ready" state by performing the following steps:

1) Removing the thread from this object's threadlist 2) Restoring the thread to its "original" owner's list

Definition at line 52 of file blocking.cpp.

The documentation for this class was generated from the following files:

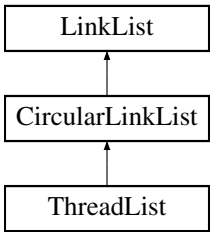
- blocking.h
- blocking.cpp

14.2 CircularLinkList Class Reference

Circular-linked-list data type, inherited from the base LinkList type.

`#include <ll.h>`

Inheritance diagram for CircularLinkList:



Public Member Functions

- virtual void Add (LinkListNode *node_)

  Add the linked list node to this linked list.
- virtual void Remove (LinkListNode *node_)

  Add the linked list node to this linked list.
- void PivotForward ()

  Pivot the head of the circularly linked list forward (Head = Head->next, Tail = Tail->next)
- void PivotBackward ()

  Pivot the head of the circularly linked list backward (Head = Head->prev, Tail = Tail->prev)

Additional Inherited Members

14.2.1 Detailed Description

Circular-linked-list data type, inherited from the base LinkList type.

Definition at line 196 of file ll.h.

14.2.2 Member Function Documentation

14.2.2.1 void CircularLinkList::Add (LinkListNode * *node_*) [virtual]

Add the linked list node to this linked list.

Parameters

*node_*	Pointer to the node to add

Implements LinkList.

Reimplemented in ThreadList.

Definition at line 102 of file ll.cpp.

14.2.2.2 void CircularLinkList::Remove (LinkListNode * *node_*) [virtual]

Add the linked list node to this linked list.

Parameters

*node_*	Pointer to the node to remove

Implements LinkList.

Reimplemented in ThreadList.

Definition at line 127 of file ll.cpp.

The documentation for this class was generated from the following files:

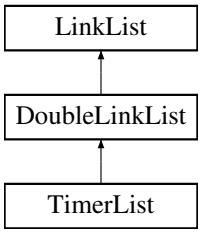
- ll.h
- ll.cpp

14.3 DoubleLinkList Class Reference

Doubly-linked-list data type, inherited from the base LinkList type.

`#include <ll.h>`

Inheritance diagram for DoubleLinkList:



Public Member Functions

- DoubleLinkList ()

  Default constructor - initializes the head/tail nodes to NULL.
- virtual void Add (LinkListNode *node_)

  Add the linked list node to this linked list.
- virtual void Remove (LinkListNode *node_)

  Add the linked list node to this linked list.

Additional Inherited Members

14.3.1 Detailed Description

Doubly-linked-list data type, inherited from the base LinkList type.

Definition at line 165 of file ll.h.

14.3.2 Member Function Documentation

14.3.2.1 void DoubleLinkList::Add (LinkListNode * *node_*) [virtual]

Add the linked list node to this linked list.

Parameters

*node_*	Pointer to the node to add

Implements LinkList.

Definition at line 41 of file ll.cpp.

14.3.2.2 void DoubleLinkList::Remove (LinkListNode * *node_*) [virtual]

Add the linked list node to this linked list.

Parameters

*node_*	Pointer to the node to remove

Implements LinkList.

Definition at line 65 of file ll.cpp.

The documentation for this class was generated from the following files:

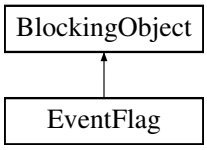
- ll.h
- ll.cpp

14.4 EventFlag Class Reference

The EventFlag class is a blocking object, similar to a semaphore or mutex, commonly used for synchronizing thread execution based on events occurring within the system.

`#include <eventflag.h>`

Inheritance diagram for EventFlag:



Public Member Functions

- void Init ()

  Init Initializes the EventFlag object prior to use.
- K_USHORT Wait (K_USHORT usMask_, EventFlagOperation_t eMode_)

  Wait - Block a thread on the specific flags in this event flag group.
- void Set (K_USHORT usMask_)

  Set - Set additional flags in this object (logical OR).
- void Clear (K_USHORT usMask_)

  ClearFlags - Clear a specific set of flags within this object, specific by bitmask.
- K_USHORT GetMask ()

  GetMask Returns the state of the 16-bit bitmask within this object.

Private Member Functions

- K_USHORT Wait_i (K_USHORT usMask_, EventFlagOperation_t eMode_)

Private Attributes

- K_USHORT **m_usSetMask**

Additional Inherited Members

14.4.1 Detailed Description

The EventFlag class is a blocking object, similar to a semaphore or mutex, commonly used for synchronizing thread execution based on events occurring within the system.

Each EventFlag object contains a 16-bit bitmask, which is used to trigger events on associated threads. Threads wishing to block, waiting for a specific event to occur can wait on any pattern within this 16-bit bitmask to be set. Here, we provide the ability for a thread to block, waiting for ANY bits in a specified mask to be set, or for ALL bits within a specific mask to be set. Depending on how the object is configured, the bits that triggered the wakeup can be automatically cleared once a match has occurred.

Definition at line 46 of file eventflag.h.

14.4.2 Member Function Documentation

14.4.2.1 void EventFlag::Clear (K_USHORT *usMask_*)

ClearFlags - Clear a specific set of flags within this object, specific by bitmask.

Parameters

*usMask_*	- Bitmask of flags to clear

Definition at line 283 of file eventflag.cpp.

14.4.2.2 K_USHORT EventFlag::GetMask ( )

GetMask Returns the state of the 16-bit bitmask within this object.

Returns

The state of the 16-bit bitmask

Definition at line 292 of file eventflag.cpp.

14.4.2.3 void EventFlag::Set (K_USHORT *usMask_*)

Set - Set additional flags in this object (logical OR).

This API can potentially result in threads blocked on Wait() to be unblocked.

Parameters

*usMask_*	- Bitmask of flags to set.

Definition at line 164 of file eventflag.cpp.

14.4.2.4 K_USHORT EventFlag::Wait (K_USHORT *usMask_*, EventFlagOperation_t *eMode_*)

Wait - Block a thread on the specific flags in this event flag group.

Parameters

*usMask_*	- 16-bit bitmask to block on
*eMode_*	- EVENT_FLAG_ANY: Thread will block on any of the bits in the mask • EVENT_FLAG_ALL: Thread will block on all of the bits in the mask

Returns

Bitmask condition that caused the thread to unblock, or 0 on error or timeout

Definition at line 146 of file eventflag.cpp.

14.4.2.5 K_USHORT EventFlag::Wait_i (K_USHORT *usMask_*, EventFlagOperation_t *eMode_*) [private]

! If the Yield operation causes a new thread to be chosen, there will ! Be a context switch at the above CS_EXIT(). The original calling ! thread will not return back until a matching SetFlags call is made ! or a timeout occurs.

Definition at line 55 of file eventflag.cpp.

The documentation for this class was generated from the following files:

- eventflag.h
- eventflag.cpp

14.5 GlobalMessagePool Class Reference

Implements a list of message objects shared between all threads.

`#include <message.h>`

Static Public Member Functions

- static void Init ()

  Initialize the message queue prior to use.
- static void Push (Message *pclMessage_)

  Return a previously-claimed message object back to the global queue.
- static Message * Pop ()

  Pop a message from the global queue, returning it to the user to be populated before sending by a transmitter.

Static Private Attributes

- static Message m_aclMessagePool [GLOBAL_MESSAGE_POOL_SIZE]

  Array of message objects that make up the message pool.
- static DoubleLinkList m_clList

  Linked list used to manage the Message objects.

14.5.1 Detailed Description

Implements a list of message objects shared between all threads.

Definition at line 157 of file message.h.

14.5.2 Member Function Documentation

14.5.2.1 Message * GlobalMessagePool::Pop ( ) [static]

Pop a message from the global queue, returning it to the user to be populated before sending by a transmitter.

Returns

Pointer to a Message object

Definition at line 70 of file message.cpp.

14.5.2.2 void GlobalMessagePool::Push (Message * *pclMessage_*) [static]

Return a previously-claimed message object back to the global queue.

Used once the message has been processed by a receiver.

Parameters

*pclMessage_*	Pointer to the Message object to return back to the global queue

Definition at line 58 of file message.cpp.

The documentation for this class was generated from the following files:

- message.h
- message.cpp

14.6 Kernel Class Reference

Class that encapsulates all of the kernel startup functions.

`#include <kernel.h>`

Static Public Member Functions

- static void Init (void)

  Kernel Initialization Function, call before any other OS function.
- static void Start (void)

  Start the kernel; function never returns.
- static bool IsStarted ()

  IsStarted.
- static void SetPanic (panic_func_t pfPanic_)

  SetPanic Set a function to be called when a kernel panic occurs, giving the user to determine the behavior when a catastrophic failure is observed.
- static bool IsPanic ()

  IsPanic Returns whether or not the kernel is in a panic state.
- static void Panic (K_USHORT usCause_)

  Panic Cause the kernel to enter its panic state.

Static Private Attributes

- static bool m_bIsStarted

  true if kernel is running, false otherwise
- static bool m_bIsPanic

  true if kernel is in panic state, false otherwise
- static panic_func_t m_pfPanic

  user-set panic function

14.6.1 Detailed Description

Class that encapsulates all of the kernel startup functions.

Definition at line 42 of file kernel.h.

14.6.2 Member Function Documentation

14.6.2.1 Kernel::Init (void) [static]

Kernel Initialization Function, call before any other OS function.

Initializes all global resources used by the operating system. This must be called before any other kernel function is invoked.

Definition at line 47 of file kernel.cpp.

14.6.2.2 static bool Kernel::IsPanic ( ) [inline], [static]

IsPanic Returns whether or not the kernel is in a panic state.

Returns

Whether or not the kernel is in a panic state

Definition at line 89 of file kernel.h.

14.6.2.3 static bool Kernel::IsStarted ( ) [inline], [static]

IsStarted.

Returns

Whether or not the kernel has started - true = running, false = not started

Definition at line 74 of file kernel.h.

14.6.2.4 void Kernel::Panic (K_USHORT *usCause_*) [static]

Panic Cause the kernel to enter its panic state.

Parameters

*usCause_*	Reason for the kernel panic

Definition at line 85 of file kernel.cpp.

14.6.2.5 static void Kernel::SetPanic (panic_func_t *pfPanic_*) [inline], [static]

SetPanic Set a function to be called when a kernel panic occurs, giving the user to determine the behavior when a catastrophic failure is observed.

Parameters

*pfPanic_*	Panic function pointer

Definition at line 83 of file kernel.h.

14.6.2.6 Kernel::Start (void) [static]

Start the kernel; function never returns.

Start the operating system kernel - the current execution context is cancelled, all kernel services are started, and the processor resumes execution at the entrypoint for the highest-priority thread.

You must have at least one thread added to the kernel before calling this function, otherwise the behavior is undefined.

Definition at line 76 of file kernel.cpp.

The documentation for this class was generated from the following files:

- kernel.h
- kernel.cpp

14.7 KernelSWI Class Reference

Class providing the software-interrupt required for context-switching in the kernel.

`#include <kernelswi.h>`

Static Public Member Functions

- static void Config (void)

  Configure the software interrupt - must be called before any other software interrupt functions are called.
- static void Start (void)

  Enable ("Start") the software interrupt functionality.
- static void Stop (void)

  Disable the software interrupt functionality.
- static void Clear (void)

  Clear the software interrupt.
- static void Trigger (void)

  Call the software interrupt.
- static K_UCHAR DI ()

  Disable the SWI flag itself.
- static void RI (bool bEnable_)

  Restore the state of the SWI to the value specified.

14.7.1 Detailed Description

Class providing the software-interrupt required for context-switching in the kernel.

Definition at line 32 of file kernelswi.h.

14.7.2 Member Function Documentation

14.7.2.1 K_UCHAR KernelSWI::DI ( ) [static]

Disable the SWI flag itself.

Returns

previous status of the SWI, prior to the DI call

Definition at line 50 of file kernelswi.cpp.

14.7.2.2 void KernelSWI::RI (bool *bEnable_*) [static]

Restore the state of the SWI to the value specified.

Parameters

*bEnable_*	true - enable the SWI, false - disable SWI

Definition at line 58 of file kernelswi.cpp.

The documentation for this class was generated from the following files:

- kernelswi.h
- kernelswi.cpp

14.8 KernelTimer Class Reference

Hardware timer interface, used by all scheduling/timer subsystems.

`#include <kerneltimer.h>`

Static Public Member Functions

- static void Config (void)

  Initializes the kernel timer before use.
- static void Start (void)

  Starts the kernel time (must be configured first)
- static void Stop (void)

  Shut down the kernel timer, used when no timers are scheduled.
- static K_UCHAR DI (void)

  Disable the kernel timer's expiry interrupt.
- static void RI (bool bEnable_)

  Retstore the state of the kernel timer's expiry interrupt.
- static void EI (void)

  Enable the kernel timer's expiry interrupt.
- static K_ULONG SubtractExpiry (K_ULONG ulInterval_)

  Subtract the specified number of ticks from the timer's expiry count register.
- static K_ULONG TimeToExpiry (void)

  Returns the number of ticks remaining before the next timer expiry.
- static K_ULONG SetExpiry (K_ULONG ulInterval_)

  Resets the kernel timer's expiry interval to the specified value.
- static K_ULONG GetOvertime (void)

  Return the number of ticks that have elapsed since the last expiry.
- static void ClearExpiry (void)

  Clear the hardware timer expiry register.

Static Private Member Functions

- static K_USHORT Read (void)

  Safely read the current value in the timer register.

14.8.1 Detailed Description

Hardware timer interface, used by all scheduling/timer subsystems.

Definition at line 33 of file kerneltimer.h.

14.8.2 Member Function Documentation

14.8.2.1 K_ULONG KernelTimer::GetOvertime (void) [static]

Return the number of ticks that have elapsed since the last expiry.

Returns

Number of ticks that have elapsed after last timer expiration

Definition at line 115 of file kerneltimer.cpp.

14.8.2.2 K_USHORT KernelTimer::Read (void) [static], [private]

Safely read the current value in the timer register.

Returns

Value held in the timer register

Definition at line 66 of file kerneltimer.cpp.

14.8.2.3 void KernelTimer::RI (bool *bEnable_*) [static]

Retstore the state of the kernel timer's expiry interrupt.

Parameters

*bEnable_*	1 enable, 0 disable

Definition at line 168 of file kerneltimer.cpp.

14.8.2.4 K_ULONG KernelTimer::SetExpiry (K_ULONG *ullnterval_*) [static]

Resets the kernel timer's expiry interval to the specified value.

Parameters

*ullnterval_*	Desired interval in ticks to set the timer for

Returns

Actual number of ticks set (may be less than desired)

Definition at line 121 of file kerneltimer.cpp.

14.8.2.5 K_ULONG KernelTimer::SubtractExpiry (K_ULONG *ullnterval_*) [static]

Subtract the specified number of ticks from the timer's expiry count register.

Returns the new expiry value stored in the register.

Parameters

*ullnterval_*	Time (in HW-specific) ticks to subtract

Returns

Value in ticks stored in the timer's expiry register

Definition at line 84 of file kerneltimer.cpp.

14.8.2.6 K_ULONG KernelTimer::TimeToExpiry (void) [static]

Returns the number of ticks remaining before the next timer expiry.

Returns

Time before next expiry in platform-specific ticks

Definition at line 95 of file kerneltimer.cpp.

The documentation for this class was generated from the following files:

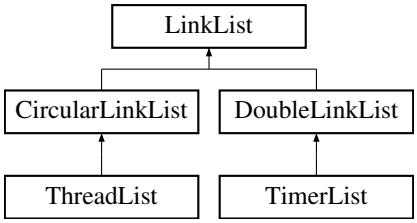
- kerneltimer.h
- kerneltimer.cpp

14.9 LinkList Class Reference

Abstract-data-type from which all other linked-lists are derived.

`#include <ll.h>`

Inheritance diagram for LinkList:



Public Member Functions

- void Init ()

  Clear the linked list.
- virtual void Add (LinkListNode *node_)=0

  Add the linked list node to this linked list.
- virtual void Remove (LinkListNode *node_)=0

  Add the linked list node to this linked list.
- LinkListNode * GetHead ()

  Get the head node in the linked list.
- LinkListNode * GetTail ()

  Get the tail node of the linked list.

Protected Attributes

- LinkListNode * m_pstHead

  Pointer to the head node in the list.
- LinkListNode * m_pstTail

  Pointer to the tail node in the list.

14.9.1 Detailed Description

Abstract-data-type from which all other linked-lists are derived.

Definition at line 112 of file ll.h.

14.9.2 Member Function Documentation

14.9.2.1 void LinkList::Add (LinkListNode * *node_*) [pure virtual]

Add the linked list node to this linked list.

Parameters

*node_*	Pointer to the node to add

Implemented in CircularLinkList, DoubleLinkList, and ThreadList.

14.9.2.2 LinkListNode * LinkList::GetHead ( ) [inline]

Get the head node in the linked list.

Returns

Pointer to the head node in the list

Definition at line 149 of file ll.h.

14.9.2.3 LinkListNode * LinkList::GetTail ( ) [inline]

Get the tail node of the linked list.

Returns

Pointer to the tail node in the list

Definition at line 158 of file ll.h.

14.9.2.4 void LinkList::Remove (LinkListNode * *node_*) [pure virtual]

Add the linked list node to this linked list.

Parameters

*node_*	Pointer to the node to remove

Implemented in CircularLinkList, DoubleLinkList, and ThreadList.

The documentation for this class was generated from the following file:

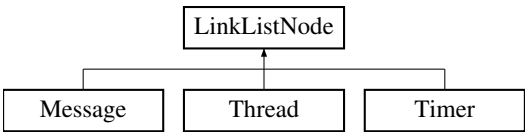
- ll.h

14.10 LinkListNode Class Reference

Basic linked-list node data structure.

`#include <ll.h>`

Inheritance diagram for LinkListNode:



Public Member Functions

- LinkListNode * GetNext (void)

Returns a pointer to the next node in the list.

- LinkListNode * GetPrev (void)

  Returns a pointer to the previous node in the list.

Protected Member Functions

- void ClearNode ()

  Initialize the linked list node, clearing its next and previous node.

Protected Attributes

- LinkListNode * next

  Pointer to the next node in the list.
- LinkListNode * prev

  Pointer to the previous node in the list.

Friends

- class **LinkList**
- class **DoubleLinkList**
- class **CircularLinkList**

14.10.1 Detailed Description

Basic linked-list node data structure.

This data is managed by the linked-list class types, and can be used transparently between them.

Definition at line 68 of file ll.h.

14.10.2 Member Function Documentation

14.10.2.1 LinkListNode * LinkListNode::GetNext (void) [inline]

Returns a pointer to the next node in the list.

Returns

a pointer to the next node in the list.

Definition at line 92 of file ll.h.

14.10.2.2 LinkListNode * LinkListNode::GetPrev (void) [inline]

Returns a pointer to the previous node in the list.

Returns

a pointer to the previous node in the list.

Definition at line 101 of file ll.h.

The documentation for this class was generated from the following files:

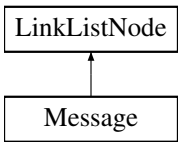
- ll.h
- ll.cpp

14.11 Message Class Reference

Class to provide message-based IPC services in the kernel.

`#include <message.h>`

Inheritance diagram for Message:



Public Member Functions

- void Init ()

  Initialize the data and code in the message.
- void SetData (void *pvData_)

  Set the data pointer for the message before transmission.
- void * GetData ()

  Get the data pointer stored in the message upon receipt.
- void SetCode (K_USHORT usCode_)

  Set the code in the message before transmission.
- K_USHORT GetCode ()

  Return the code set in the message upon receipt.

Private Attributes

- void * m_pvData

  Pointer to the message data.
- K_USHORT m_usCode

  Message code, providing context for the message.

Additional Inherited Members

14.11.1 Detailed Description

Class to provide message-based IPC services in the kernel.

Definition at line 99 of file message.h.

14.11.2 Member Function Documentation

14.11.2.1 K_USHORT Message::GetCode ( ) [inline]

Return the code set in the message upon receipt.

Returns

User code set in the object

Definition at line 143 of file message.h.

14.11.2.2 void * Message::GetData ( ) [inline]

Get the data pointer stored in the message upon receipt.

Returns

Pointer to the data set in the message object

Definition at line 125 of file message.h.

14.11.2.3 Message::SetCode (K_USHORT *usCode_*) [inline]

Set the code in the message before transmission.

Parameters

*usCode_*	Data code to set in the object

Definition at line 134 of file message.h.

14.11.2.4 void Message::SetData (void * *pvData_*) [inline]

Set the data pointer for the message before transmission.

Parameters

*pvData_*	Pointer to the data object to send in the message

Definition at line 116 of file message.h.

The documentation for this class was generated from the following file:

- message.h

14.12 MessageQueue Class Reference

List of messages, used as the channel for sending and receiving messages between threads.

`#include <message.h>`

Public Member Functions

- void Init ()

  Initialize the message queue prior to use.
- Message * Receive ()

  Receive a message from the message queue.
- void Send (Message *pclSrc_)

  Send a message object into this message queue.
- K_USHORT GetCount ()

  Return the number of messages pending in the "receive" queue.

Private Member Functions

- Message * **Receive_i** (void)

Private Attributes

- Semaphore m_clSemaphore

  Counting semaphore used to manage thread blocking.
- DoubleLinkList m_clLinkList

  List object used to store messages.

14.12.1 Detailed Description

List of messages, used as the channel for sending and receiving messages between threads.

Definition at line 201 of file message.h.

14.12.2 Member Function Documentation

14.12.2.1 K_USHORT MessageQueue::GetCount ( )

Return the number of messages pending in the "receive" queue.

Returns

Count of pending messages in the queue.

Definition at line 156 of file message.cpp.

14.12.2.2 Message * MessageQueue::Receive ( )

Receive a message from the message queue.

If the message queue is empty, the thread will block until a message is available.

Returns

Pointer to a message object at the head of the queue

Definition at line 92 of file message.cpp.

14.12.2.3 void MessageQueue::Send (Message * *pclSrc_*)

Send a message object into this message queue.

Will un-block the first waiting thread blocked on this queue if that occurs.

Parameters

*pclSrc_*	Pointer to the message object to add to the queue

Definition at line 140 of file message.cpp.

The documentation for this class was generated from the following files:

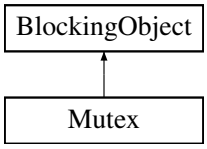
- message.h
- message.cpp

14.13 Mutex Class Reference

Mutual-exclusion locks, based on BlockingObject.

`#include <mutex.h>`

Inheritance diagram for Mutex:



Public Member Functions

- void Init ()

  Initialize a mutex object for use - must call this function before using the object.
- void Claim ()

  Claim the mutex.
- void Release ()

  Release the mutex.

Private Member Functions

- K_UCHAR WakeNext ()

  Wake the next thread waiting on the Mutex.
- void **Claim_i** (void)

Private Attributes

- K_UCHAR m_ucRecurse

  The recursive lock-count when a mutex is claimed multiple times by the same owner.
- K_UCHAR m_bReady

  State of the mutex - true = ready, false = claimed.
- K_UCHAR m_ucMaxPri

  Maximum priority of thread in queue, used for priority inheritence.
- Thread * m_pclOwner

  Pointer to the thread that owns the mutex (when claimed)

Additional Inherited Members

14.13.1 Detailed Description

Mutual-exclusion locks, based on BlockingObject.

Definition at line 68 of file mutex.h.

14.13.2 Member Function Documentation

14.13.2.1 void Mutex::Claim (void)

Claim the mutex.

When the mutex is claimed, no other thread can claim a region protected by the object.

Definition at line 199 of file mutex.cpp.

14.13.2.2 void Mutex::Release ( )

Release the mutex.

When the mutex is released, another object can enter the mutex-protected region.

Definition at line 217 of file mutex.cpp.

The documentation for this class was generated from the following files:

- mutex.h
- mutex.cpp

14.14 Quantum Class Reference

Static-class used to implement Thread quantum functionality, which is a key part of round-robin scheduling.

`#include <quantum.h>`

Static Public Member Functions

- static void UpdateTimer ()

  This function is called to update the thread quantum timer whenever something in the scheduler has changed.
- static void AddThread (Thread *pclThread_)

  Add the thread to the quantum timer.
- static void RemoveThread ()

  Remove the thread from the quantum timer.
- static void SetInTimer (void)

  SetInTimer.
- static void ClearInTimer (void)

  ClearInTimer.

Static Private Member Functions

- static void SetTimer (Thread *pclThread_)

  Set up the quantum timer in the timer scheduler.

Static Private Attributes

- static Timer **m_clQuantumTimer**
- static K_UCHAR **m_bActive**
- static K_UCHAR **m_bInTimer**

14.14.1 Detailed Description

Static-class used to implement Thread quantum functionality, which is a key part of round-robin scheduling.

Definition at line 39 of file quantum.h.

14.14.2 Member Function Documentation

14.14.2.1 void Quantum::AddThread (Thread * *pclThread_*) [static]

Add the thread to the quantum timer.

Only one thread can own the quantum, since only one thread can be running on a core at a time.

Definition at line 71 of file quantum.cpp.

14.14.2.2 static void Quantum::ClearInTimer (void) [inline], [static]

ClearInTimer.

Clear the flag once the timer callback function has been completed.

Definition at line 82 of file quantum.h.

14.14.2.3 void Quantum::RemoveThread (void) [static]

Remove the thread from the quantum timer.

This will cancel the timer.

Definition at line 97 of file quantum.cpp.

14.14.2.4 static void Quantum::SetInTimer (void) [inline], [static]

SetInTimer.

Set a flag to indicate that the CPU is currently running within the timer-callback routine. This prevents the Quantum timer from being updated in the middle of a callback cycle, potentially resulting in the kernel timer becoming disabled.

Definition at line 75 of file quantum.h.

14.14.2.5 void Quantum::SetTimer (Thread * *pclThread_*) [static], [private]

Set up the quantum timer in the timer scheduler.

This creates a one-shot timer, which calls a static callback in quantum.cpp that on expiry will pivot the head of the threadlist for the thread's priority. This is the mechanism that provides round-robin scheduling in the system.

Parameters

*pclThread_*	Pointer to the thread to set the Quantum timer on

Definition at line 61 of file quantum.cpp.

14.14.2.6 void Quantum::UpdateTimer (void) [static]

This function is called to update the thread quantum timer whenever something in the scheduler has changed.

This can result in the timer being re-loaded or started. The timer is never stopped, but if may be ignored on expiry.

Definition at line 110 of file quantum.cpp.

The documentation for this class was generated from the following files:

- quantum.h
- quantum.cpp

14.15 Scheduler Class Reference

Priority-based round-robin Thread scheduling, using ThreadLists for housekeeping.

`#include <scheduler.h>`

Static Public Member Functions

- static void Init ()

  Intiailize the scheduler, must be called before use.
- static void Schedule ()

  Run the scheduler, determines the next thread to run based on the current state of the threads.
- static void Add (Thread *pclThread_)

  Add a thread to the scheduler at its current priority level.
- static void Remove (Thread *pclThread_)

  Remove a thread from the scheduler at its current priority level.
- static K_BOOL SetScheduler (K_BOOL bEnable_)

  Set the active state of the scheduler.
- static Thread * GetCurrentThread ()

  Return the pointer to the currently-running thread.
- static Thread * GetNextThread ()

  Return the pointer to the thread that should run next, according to the last run of the scheduler.
- static ThreadList * GetThreadList (K_UCHAR ucPriority_)

  Return the pointer to the active list of threads that are at the given priority level in the scheduler.
- static ThreadList * GetStopList ()

  Return the pointer to the list of threads that are in the scheduler's stopped state.
- static K_UCHAR IsEnabled ()

  Return the current state of the scheduler - whether or not scheudling is enabled or disabled.
- static void **QueueScheduler** ()

Static Private Attributes

- static K_BOOL m_bEnabled

  Scheduler's state - enabled or disabled.
- static K_BOOL m_bQueuedSchedule

  Variable representing whether or not there's a queued scheduler operation.
- static ThreadList m_clStopList

  ThreadList for all stopped threads.
- static ThreadList m_aclPriorities [NUM_PRIORITIES]

  ThreadLists for all threads at all priorities.
- static K_UCHAR m_ucPriFlag

  Bitmap flag for each.

14.15.1 Detailed Description

Priority-based round-robin Thread scheduling, using ThreadLists for housekeeping.

Definition at line 62 of file scheduler.h.

14.15.2 Member Function Documentation

14.15.2.1 void Scheduler::Add (Thread * *pclThread_*) [static]

Add a thread to the scheduler at its current priority level.

Parameters

*pclThread_*	Pointer to the thread to add to the scheduler

Definition at line 81 of file scheduler.cpp.

14.15.2.2 static Thread* Scheduler::GetCurrentThread ( ) [inline], [static]

Return the pointer to the currently-running thread.

Returns

Pointer to the currently-running thread

Definition at line 119 of file scheduler.h.

14.15.2.3 static Thread* Scheduler::GetNextThread ( ) [inline], [static]

Return the pointer to the thread that should run next, according to the last run of the scheduler.

Returns

Pointer to the next-running thread

Definition at line 127 of file scheduler.h.

14.15.2.4 static ThreadList* Scheduler::GetStopList ( ) [inline], [static]

Return the pointer to the list of threads that are in the scheduler's stopped state.

Returns

Pointer to the ThreadList containing the stopped threads

Definition at line 145 of file scheduler.h.

14.15.2.5 static ThreadList* Scheduler::GetThreadList (K_UCHAR *ucPriority_*) [inline], [static]

Return the pointer to the active list of threads that are at the given priority level in the scheduler.

Parameters

*ucPriority_*	Priority level of

Returns

Pointer to the ThreadList for the given priority level

Definition at line 137 of file scheduler.h.

14.15.2.6 K_UCHAR Scheduler::IsEnabled ( ) [inline], [static]

Return the current state of the scheduler - whether or not scheudling is enabled or disabled.

Returns

true - scheduler enabled, false - disabled

Definition at line 155 of file scheduler.h.

14.15.2.7 void Scheduler::Remove (Thread * *pclThread_*) [static]

Remove a thread from the scheduler at its current priority level.

Parameters

*pclThread_*	Pointer to the thread to be removed from the scheduler

Definition at line 88 of file scheduler.cpp.

14.15.2.8 Scheduler::Schedule ( ) [static]

Run the scheduler, determines the next thread to run based on the current state of the threads.

Note that the next-thread chosen from this function is only valid while in a critical section.

Definition at line 64 of file scheduler.cpp.

14.15.2.9 void Scheduler::SetScheduler (K_BOOL *bEnable_*) [static]

Set the active state of the scheduler.

When the scheduler is disabled, the *next thread* is never set; the currently running thread will run forever until the scheduler is enabled again. Care must be taken to ensure that we don't end up trying to block while the scheduler is disabled, otherwise the system ends up in an unusable state.

Parameters

*bEnable_*	true to enable, false to disable the scheduler

Definition at line 95 of file scheduler.cpp.

The documentation for this class was generated from the following files:

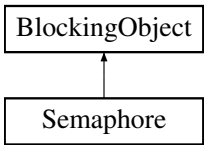
- scheduler.h
- scheduler.cpp

14.16 Semaphore Class Reference

Counting semaphore, based on BlockingObject base class.

`#include <ksemaphore.h>`

Inheritance diagram for Semaphore:



Public Member Functions

- void Init (K_USHORT usInitVal_, K_USHORT usMaxVal_)

  Initialize a semaphore before use.
- bool Post ()

  Increment the semaphore count.
- void Pend ()

  Decrement the semaphore count.
- K_USHORT GetCount ()

  Return the current semaphore counter.

Private Member Functions

- K_UCHAR WakeNext ()

  Wake the next thread waiting on the semaphore.
- void **Pend_i** (void)

Private Attributes

- K_USHORT **m_usValue**
- K_USHORT **m_usMaxValue**

Additional Inherited Members

14.16.1 Detailed Description

Counting semaphore, based on BlockingObject base class.

Definition at line 37 of file ksemaphore.h.

14.16.2 Member Function Documentation

14.16.2.1 K_USHORT Semaphore::GetCount ( )

Return the current semaphore counter.

This can be used by a thread to bypass blocking on a semaphore - allowing it to do other things until a non-zero count is returned, instead of blocking until the semaphore is posted.

Returns

The current semaphore counter value.

Definition at line 223 of file ksemaphore.cpp.

14.16.2.2 void Semaphore::Init (K_USHORT *usInitVal_*, K_USHORT *usMaxVal_*)

Initialize a semaphore before use.

Must be called before post/pend operations.

Parameters

*usInitVal_*	Initial value held by the semaphore
*usMaxVal_*	Maximum value for the semaphore

Definition at line 84 of file ksemaphore.cpp.

14.16.2.3 void Semaphore::Pend ( )

Decrement the semaphore count.

If the count is zero, the thread will block until the semaphore is pended.

Definition at line 205 of file ksemaphore.cpp.

14.16.2.4 void Semaphore::Post ( )

Increment the semaphore count.

Returns

true if the semaphore was posted, false if the count is already maxed out.

Definition at line 96 of file ksemaphore.cpp.

The documentation for this class was generated from the following files:

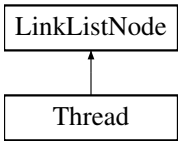
- ksemaphore.h
- ksemaphore.cpp

14.17 Thread Class Reference

Object providing fundamental multitasking support in the kernel.

`#include <thread.h>`

Inheritance diagram for Thread:



Public Member Functions

- void Init (K_WORD *paucStack_, K_USHORT usStackSize_, K_UCHAR ucPriority_, ThreadEntry_t pfEntryPoint_, void *pvArg_)

  Initialize a thread prior to its use.
- void Start ()

  Start the thread - remove it from the stopped list, add it to the scheduler's list of threads (at the thread's set priority), and continue along.
- void Stop ()

  Stop a thread that's actively scheduled without destroying its stacks.
- void SetName (const K_CHAR *szName_)

  Set the name of the thread - this is purely optional, but can be useful when identifying issues that come along when multiple threads are at play in a system.
- const K_CHAR * GetName ()
- ThreadList * GetOwner (void)

  Return the ThreadList where the thread belongs when it's in the active/ready state in the scheduler.
- ThreadList * GetCurrent (void)

  Return the ThreadList where the thread is currently located.
- K_UCHAR GetPriority (void)

  Return the priority of the current thread.
- K_UCHAR GetCurPriority (void)

  Return the priority of the current thread.
- void SetQuantum (K_USHORT usQuantum_)

  Set the thread's round-robin execution quantum.
- K_USHORT GetQuantum (void)

  Get the thread's round-robin execution quantum.
- void SetCurrent (ThreadList *pclNewList_)

  Set the thread's current to the specified thread list.
- void SetOwner (ThreadList *pclNewList_)

  Set the thread's owner to the specified thread list.
- void SetPriority (K_UCHAR ucPriority_)

  Set the priority of the Thread (running or otherwise) to a different level.
- void InheritPriority (K_UCHAR ucPriority_)

  Allow the thread to run at a different priority level (temporarily) for the purpose of avoiding priority inversions.
- void Exit ()

  Remove the thread from being scheduled again.
- void SetID (K_UCHAR ucID_)

  Set an 8-bit ID to uniquely identify this thread.
- K_UCHAR GetID ()

  Return the 8-bit ID corresponding to this thread.
- K_USHORT GetStackSlack ()

  Performs a (somewhat lengthy) check on the thread stack to check the amount of stack margin (or "slack") remaining on the stack.
- K_USHORT GetEventFlagMask ()

  GetEventFlagMask returns the thread's current event-flag mask, which is used in conjunction with the EventFlag blocking object type.
- void SetEventFlagMask (K_USHORT usMask_)

  SetEventFlagMask Sets the active event flag bitfield mask.
- void SetEventFlagMode (EventFlagOperation_t eMode_)

  SetEventFlagMode Sets the active event flag operation mode.
- EventFlagOperation_t GetEventFlagMode ()

  GetEventFlagMode Returns the thread's event flag's operating mode.

- Timer * GetTimer ()

  Return a pointer to the thread's timer object.
- void **SetExpired** (K_BOOL bExpired_)
- K_BOOL **GetExpired** ()

Static Public Member Functions

- static void Sleep (K_ULONG ulTimeMs_)

  Put the thread to sleep for the specified time (in milliseconds).
- static void USleep (K_ULONG ulTimeUs_)

  Put the thread to sleep for the specified time (in microseconds).
- static void Yield (void)

  Yield the thread - this forces the system to call the scheduler and determine what thread should run next.

Private Member Functions

- void SetPriorityBase (K_UCHAR ucPriority_)

Static Private Member Functions

- static void ContextSwitchSWI (void)

  This code is used to trigger the context switch interrupt.

Private Attributes

- K_WORD * m_pwStackTop

  Pointer to the top of the thread's stack.
- K_WORD * m_pwStack

  Pointer to the thread's stack.
- K_USHORT m_usStackSize

  Size of the stack (in bytes)
- K_USHORT m_usQuantum

  Thread quantum (in milliseconds)
- K_UCHAR m_ucThreadID

  Thread ID.
- K_UCHAR m_ucPriority

  Default priority of the thread.
- K_UCHAR m_ucCurPriority

  Current priority of the thread (priority inheritence)
- ThreadEntry_t m_pfEntryPoint

  The entry-point function called when the thread starts.
- void * m_pvArg

  Pointer to the argument passed into the thread's entrypoint.
- const K_CHAR * m_szName

  Thread name.
- K_USHORT m_usFlagMask

  Event-flag mask.
- EventFlagOperation_t m_eFlagMode

  Event-flag mode.
- Timer m_clTimer

Timer used for blocking-object timeouts.

- K_BOOL **m_bExpired**
- ThreadList * m_pclCurrent

  Pointer to the thread-list where the thread currently resides.
- ThreadList * m_pclOwner

  Pointer to the thread-list where the thread resides when active.

Friends

- class **ThreadPort**

Additional Inherited Members

14.17.1 Detailed Description

Object providing fundamental multitasking support in the kernel.

Definition at line 57 of file thread.h.

14.17.2 Member Function Documentation

14.17.2.1 void Thread::ContextSwitchSWI (void) [static], [private]

This code is used to trigger the context switch interrupt.

Called whenever the kernel decides that it is necessary to swap out the current thread for the "next" thread.

Definition at line 353 of file thread.cpp.

14.17.2.2 void Thread::Exit ( )

Remove the thread from being scheduled again.

The thread is effectively destroyed when this occurs. This is extremely useful for cases where a thread encounters an unrecoverable error and needs to be restarted, or in the context of systems where threads need to be created and destroyed dynamically.

This must not be called on the idle thread.

Definition at line 151 of file thread.cpp.

14.17.2.3 K_UCHAR Thread::GetCurPriority (void) [inline]

Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 160 of file thread.h.

14.17.2.4 ThreadList * Thread::GetCurrent (void) [inline]

Return the ThreadList where the thread is currently located.

Returns

Pointer to the thread's current list

Definition at line 141 of file thread.h.

14.17.2.5 K_USHORT Thread::GetEventFlagMask ( ) [inline]

GetEventFlagMask returns the thread's current event-flag mask, which is used in conjunction with the EventFlag blocking object type.

Returns

A copy of the thread's event flag mask

Definition at line 313 of file thread.h.

14.17.2.6 EventFlagOperation_t Thread::GetEventFlagMode ( ) [inline]

GetEventFlagMode Returns the thread's event flag's operating mode.

Returns

The thread's event flag mode.

Definition at line 332 of file thread.h.

14.17.2.7 K_UCHAR Thread::GetID ( ) [inline]

Return the 8-bit ID corresponding to this thread.

Returns

Thread's 8-bit ID, set by the user

Definition at line 288 of file thread.h.

14.17.2.8 const K_CHAR * Thread::GetName ( ) [inline]

Returns

Pointer to the name of the thread. If this is not set, will be NULL.

Definition at line 121 of file thread.h.

14.17.2.9 ThreadList * Thread::GetOwner (void) [inline]

Return the ThreadList where the thread belongs when it's in the active/ready state in the scheduler.

Returns

Pointer to the Thread's owner list

Definition at line 132 of file thread.h.

14.17.2.10 K_UCHAR Thread::GetPriority (void) [inline]

Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 151 of file thread.h.

14.17.2.11 K_USHORT Thread::GetQuantum (void) [inline]

Get the thread's round-robin execution quantum.

Returns

The thread's quantum

Definition at line 179 of file thread.h.

14.17.2.12 K_USHORT Thread::GetStackSlack ( )

Performs a (somewhat lengthy) check on the thread stack to check the amount of stack margin (or "slack") remaining on the stack.

If you're having problems with blowing your stack, you can run this function at points in your code during development to see what operations cause problems. Also useful during development as a tool to optimally size thread stacks.

Returns

The amount of slack (unused bytes) on the stack

! ToDo: Take into account stacks that grow up

Definition at line 242 of file thread.cpp.

14.17.2.13 void Thread::InheritPriority (K_UCHAR *ucPriority_*)

Allow the thread to run at a different priority level (temporarily) for the purpose of avoiding priority inversions.

This should only be called from within the implementation of blocking-objects.

Parameters

*ucPriority_*	New Priority to boost to.

Definition at line 346 of file thread.cpp.

14.17.2.14 void Thread::Init (K_WORD * *paucStack_*, K_USHORT *usStackSize_*, K_UCHAR *ucPriority_*, ThreadEntry_t *pfEntryPoint_*, void * *pvArg_*)

Initialize a thread prior to its use.

Initialized threads are placed in the stopped state, and are not scheduled until the thread's start method has been invoked first.

Parameters

*paucStack_*	Pointer to the stack to use for the thread
*usStackSize_*	Size of the stack (in bytes)
*ucPriority_*	Priority of the thread (0 = idle, 7 = max)
*pfEntryPoint_*	This is the function that gets called when the thread is started
*pvArg_*	Pointer to the argument passed into the thread's entrypoint function.

< Default round-robin thread quantum of 4ms

Definition at line 41 of file thread.cpp.

14.17.2.15 void Thread::SetCurrent (ThreadList * *pclNewList_*) [inline]

Set the thread's current to the specified thread list.

Parameters

*pclNewList_*	Pointer to the threadlist to apply thread ownership

Definition at line 189 of file thread.h.

14.17.2.16 void Thread::SetEventFlagMask (K_USHORT *usMask_*) [inline]

SetEventFlagMask Sets the active event flag bitfield mask.

Parameters

*usMask_*	

Definition at line 319 of file thread.h.

14.17.2.17 void Thread::SetEventFlagMode (EventFlagOperation_t *eMode_*) [inline]

SetEventFlagMode Sets the active event flag operation mode.

Parameters

*eMode_*	Event flag operation mode, defines the logical operator to apply to the event flag.

Definition at line 326 of file thread.h.

14.17.2.18 void Thread::SetID (K_UCHAR *ucID_*) [inline]

Set an 8-bit ID to uniquely identify this thread.

Parameters

*ucID_*	8-bit Thread ID, set by the user

Definition at line 279 of file thread.h.

14.17.2.19 void Thread::SetName (const K_CHAR * *szName_*) [inline]

Set the name of the thread - this is purely optional, but can be useful when identifying issues that come along when multiple threads are at play in a system.

Parameters

*szName_*	Char string containing the thread name

Definition at line 113 of file thread.h.

14.17.2.20 void Thread::SetOwner (ThreadList * *pclNewList_*) [inline]

Set the thread's owner to the specified thread list.

Parameters

*pclNewList_*	Pointer to the threadlist to apply thread ownership

Definition at line 198 of file thread.h.

14.17.2.21 void Thread::SetPriority (K_UCHAR *ucPriority_*)

Set the priority of the Thread (running or otherwise) to a different level.

This activity involves re-scheduling, and must be done so with due caution, as it may effect the determinism of the system.

This should *always* be called from within a critical section to prevent system issues.

Parameters

*ucPriority_*	New priority of the thread

Definition at line 303 of file thread.cpp.

14.17.2.22 void Thread::SetPriorityBase (K_UCHAR *ucPriority_*) [private]

Parameters

*ucPriority_*	

Definition at line 293 of file thread.cpp.

14.17.2.23 void Thread::SetQuantum (K_USHORT *usQuantum_*) [inline]

Set the thread's round-robin execution quantum.

Parameters

*usQuantum_*	Thread's execution quantum (in milliseconds)

Definition at line 170 of file thread.h.

14.17.2.24 void Thread::Sleep (K_ULONG *ulTimeMs_*) [static]

Put the thread to sleep for the specified time (in milliseconds).

Actual time slept may be longer (but not less than) the interval specified.

Parameters

*ulTimeMs_*	Time to sleep (in ms)

Definition at line 197 of file thread.cpp.

14.17.2.25 void Thread::Stop (void)

Stop a thread that's actively scheduled without destroying its stacks.

Stopped threads can be restarted using the Start() API.

Definition at line 123 of file thread.cpp.

14.17.2.26 void Thread::USleep (K_ULONG *ulTimeUs_*) [static]

Put the thread to sleep for the specified time (in microseconds).

Actual time slept may be longer (but not less than) the interval specified.

Parameters

*ulTimeUs_*	Time to sleep (in microseconds)

Definition at line 219 of file thread.cpp.

14.17.2.27 void Thread::Yield (void) [static]

Yield the thread - this forces the system to call the scheduler and determine what thread should run next.

This is typically used when threads are moved in and out of the scheduler.

Definition at line 263 of file thread.cpp.

The documentation for this class was generated from the following files:

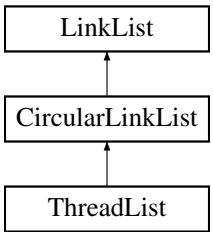
- thread.h
- thread.cpp

14.18 ThreadList Class Reference

This class is used for building thread-management facilities, such as schedulers, and blocking objects.

`#include <threadlist.h>`

Inheritance diagram for ThreadList:



Public Member Functions

- ThreadList ()

  Default constructor - zero-initializes the data.
- void SetPriority (K_UCHAR ucPriority_)

  Set the priority of this threadlist (if used for a scheduler).
- void SetFlagPointer (K_UCHAR *pucFlag_)

  Set the pointer to a bitmap to use for this threadlist.

- void Add (LinkListNode *node_)

  Add a thread to the threadlist.
- void Add (LinkListNode *node_, K_UCHAR *pucFlag_, K_UCHAR ucPriority_)

  Add a thread to the threadlist, specifying the flag and priority at the same time.
- void Remove (LinkListNode *node_)

  Remove the specified thread from the threadlist.
- Thread * HighestWaiter ()

  Return a pointer to the highest-priority thread in the thread-list.

Private Attributes

- K_UCHAR m_ucPriority

  Priority of the threadlist.
- K_UCHAR * m_pucFlag

  Pointer to the bitmap/flag to set when used for scheduling.

Additional Inherited Members

14.18.1 Detailed Description

This class is used for building thread-management facilities, such as schedulers, and blocking objects.

Definition at line 34 of file threadlist.h.

14.18.2 Member Function Documentation

14.18.2.1 void ThreadList::Add (LinkListNode * *node_*) [virtual]

Add a thread to the threadlist.

Parameters

*node_*	Pointer to the thread (link list node) to add to the list

Reimplemented from CircularLinkList.

Definition at line 46 of file threadlist.cpp.

14.18.2.2 void ThreadList::Add (LinkListNode * *node_*, K_UCHAR * *pucFlag_*, K_UCHAR *ucPriority_*)

Add a thread to the threadlist, specifying the flag and priority at the same time.

Parameters

*node_*	Pointer to the thread to add (link list node)
*pucFlag_*	Pointer to the bitmap flag to set (if used in a scheduler context), or NULL for non-scheduler.
*ucPriority_*	Priority of the threadlist

Definition at line 62 of file threadlist.cpp.

14.18.2.3 Thread * ThreadList::HighestWaiter ( )

Return a pointer to the highest-priority thread in the thread-list.

Returns

Pointer to the highest-priority thread

Definition at line 87 of file threadlist.cpp.

14.18.2.4 void ThreadList::Remove (LinkListNode * *node_*) [virtual]

Remove the specified thread from the threadlist.

Parameters

*node_*	Pointer to the thread to remove

Reimplemented from CircularLinkList.

Definition at line 71 of file threadlist.cpp.

14.18.2.5 void ThreadList::SetFlagPointer (K_UCHAR * *pucFlag_*)

Set the pointer to a bitmap to use for this threadlist.

Once again, only needed when the threadlist is being used for scheduling purposes.

Parameters

*pucFlag_*	Pointer to the bitmap flag

Definition at line 40 of file threadlist.cpp.

14.18.2.6 void ThreadList::SetPriority (K_UCHAR *ucPriority_*)

Set the priority of this threadlist (if used for a scheduler).

Parameters

*ucPriority_*	Priority level of the thread list

Definition at line 34 of file threadlist.cpp.

The documentation for this class was generated from the following files:

- threadlist.h
- threadlist.cpp

14.19 ThreadPort Class Reference

Class defining the architecture specific functions required by the kernel.

`#include <threadport.h>`

Static Public Member Functions

- static void StartThreads ()

  Function to start the scheduler, initial threads, etc.

Static Private Member Functions

- static void InitStack (Thread *pstThread_)

  Initialize the thread's stack.

Friends

- class **Thread**

14.19.1 Detailed Description

Class defining the architecture specific functions required by the kernel.

This is limited (at this point) to a function to start the scheduler, and a function to initialize the default stack-frame for a thread.

Definition at line 167 of file threadport.h.

14.19.2 Member Function Documentation

14.19.2.1 void ThreadPort::InitStack (Thread * *pstThread_*) [static], [private]

Initialize the thread's stack.

Parameters

*pstThread_*	Pointer to the thread to initialize

Definition at line 37 of file threadport.cpp.

The documentation for this class was generated from the following files:

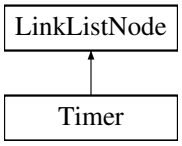
- threadport.h
- threadport.cpp

14.20 Timer Class Reference

Timer - an event-driven execution context based on a specified time interval.

`#include <timerlist.h>`

Inheritance diagram for Timer:



Public Member Functions

- Timer ()

  Default Constructor - zero-initializes all internal data.
- void Init ()

  Re-initialize the Timer to default values.

- void Start (bool bRepeat_, K_ULONG ulIntervalMs_, TimerCallback_t pfCallback_, void *pvData_)

  Start a timer using default ownership, using repeats as an option, and millisecond resolution.
- void Start (bool bRepeat_, K_ULONG ulIntervalMs_, K_ULONG ulToleranceMs_, TimerCallback_t pfCallback_, void *pvData_)

  Start a timer using default ownership, using repeats as an option, and millisecond resolution.
- void Stop ()

  Stop a timer already in progress.
- void SetFlags (K_UCHAR ucFlags_)

  Set the timer's flags based on the bits in the ucFlags_ argument.
- void SetCallback (TimerCallback_t pfCallback_)

  Define the callback function to be executed on expiry of the timer.
- void SetData (void *pvData_)

  Define a pointer to be sent to the timer callbcak on timer expiry.
- void SetOwner (Thread *pclOwner_)

  Set the owner-thread of this timer object (all timers must be owned by a thread).
- void SetIntervalTicks (K_ULONG ulTicks_)

  Set the timer expiry in system-ticks (platform specific!)
- void SetIntervalSeconds (K_ULONG ulSeconds_)

  ! The next three cost us 330 bytes of flash on AVR...
- K_ULONG **GetInterval** ()
- void SetIntervalMSeconds (K_ULONG ulMSeconds_)

  Set the timer expiry interval in milliseconds (platform agnostic)
- void SetIntervalUSeconds (K_ULONG ulUSeconds_)

  Set the timer expiry interval in microseconds (platform agnostic)
- void SetTolerance (K_ULONG ulTicks_)

  Set the timer's maximum tolerance in order to synchronize timer processing with other timers in the system.

Private Attributes

- K_UCHAR m_ucFlags

  Flags for the timer, defining if the timer is one-shot or repeated.
- TimerCallback_t m_pfCallback

  Pointer to the callback function.
- K_ULONG m_ulInterval

  Interval of the timer in timer ticks.
- K_ULONG m_ulTimeLeft

  Time remaining on the timer.
- K_ULONG m_ulTimerTolerance

  Maximum tolerance (used for timer harmonization)
- Thread * m_pclOwner

  Pointer to the owner thread.
- void * m_pvData

  Pointer to the callback data.

Friends

- class **TimerList**

Additional Inherited Members

14.20.1 Detailed Description

Timer - an event-driven execution context based on a specified time interval.

This inherits from a LinkListNode for ease of management by a global TimerList object.

Definition at line 99 of file timerlist.h.

14.20.2 Member Function Documentation

14.20.2.1 void Timer::SetCallback (TimerCallback_t *pfCallback_*) [inline]

Define the callback function to be executed on expiry of the timer.

Parameters

*pfCallback_*	Pointer to the callback function to call

Definition at line 160 of file timerlist.h.

14.20.2.2 void Timer::SetData (void * *pvData_*) [inline]

Define a pointer to be sent to the timer callbcak on timer expiry.

Parameters

*pvData_*	Pointer to data to pass as argument into the callback

Definition at line 169 of file timerlist.h.

14.20.2.3 void Timer::SetFlags (K_UCHAR *ucFlags_*) [inline]

Set the timer's flags based on the bits in the ucFlags_ argument.

Parameters

*ucFlags_*	Flags to assign to the timer object. TIMERLIST_FLAG_ONE_SHOT for a one-shot timer, 0 for a continuous timer.

Definition at line 151 of file timerlist.h.

14.20.2.4 void Timer::SetIntervalMSeconds (K_ULONG *ulMSeconds_*)

Set the timer expiry interval in milliseconds (platform agnostic)

Parameters

*ulMSeconds_*	Time in milliseconds

Definition at line 304 of file timerlist.cpp.

14.20.2.5 void Timer::SetIntervalSeconds (K_ULONG *ulSeconds_*)

! The next three cost us 330 bytes of flash on AVR...

Set the timer expiry interval in seconds (platform agnostic)

Parameters

*ulSeconds_*	Time in seconds

Definition at line 298 of file timerlist.cpp.

14.20.2.6 void Timer::SetIntervalTicks (K_ULONG *ulTicks_*)

Set the timer expiry in system-ticks (platform specific!)

Parameters

*ulTicks_*	Time in ticks

Definition at line 290 of file timerlist.cpp.

14.20.2.7 void Timer::SetIntervalUSeconds (K_ULONG *ulUSeconds_*)

Set the timer expiry interval in microseconds (platform agnostic)

Parameters

*ulUSeconds_*	Time in microseconds

Definition at line 310 of file timerlist.cpp.

14.20.2.8 void Timer::SetOwner (Thread * *pclOwner_*) [inline]

Set the owner-thread of this timer object (all timers must be owned by a thread).

Parameters

*pclOwner_*	Owner thread of this timer object

Definition at line 179 of file timerlist.h.

14.20.2.9 void Timer::SetTolerance (K_ULONG *ulTicks_*)

Set the timer's maximum tolerance in order to synchronize timer processing with other timers in the system.

Parameters

*ulTicks_*	Maximum tolerance in ticks

Definition at line 316 of file timerlist.cpp.

14.20.2.10 void Timer::Start (bool *bRepeat_*, K_ULONG *ulIntervalMs_*, TimerCallback_t *pfCallback_*, void * *pvData_*)

Start a timer using default ownership, using repeats as an option, and millisecond resolution.

Parameters

*bRepeat_*	0 - timer is one-shot. 1 - timer is repeating.
*ulIntervalMs_*	- Interval of the timer in miliseconds
*pfCallback_*	- Function to call on timer expiry
*pvData_*	- Data to pass into the callback function

Definition at line 259 of file timerlist.cpp.

14.20.2.11 void Timer::Start (bool *bRepeat_*, K_ULONG *ulIntervalMs_*, K_ULONG *ulToleranceMs_*, TimerCallback_t *pfCallback_*, void * *pvData_*)

Start a timer using default ownership, using repeats as an option, and millisecond resolution.

Parameters

*bRepeat_*	0 - timer is one-shot. 1 - timer is repeating.
*ulIntervalMs_*	- Interval of the timer in miliseconds
ulToleranceMs	- Allow the timer expiry to be delayed by an additional maximum time, in order to have as many timers expire at the same time as possible.
*pfCallback_*	- Function to call on timer expiry
*pvData_*	- Data to pass into the callback function

Definition at line 277 of file timerlist.cpp.

14.20.2.12 void Timer::Stop (void)

Stop a timer already in progress.

Has no effect on timers that have already been stopped.

Definition at line 284 of file timerlist.cpp.

The documentation for this class was generated from the following files:

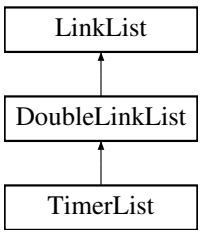
- timerlist.h
- timerlist.cpp

14.21 TimerList Class Reference

TimerList class - a doubly-linked-list of timer objects.

`#include <timerlist.h>`

Inheritance diagram for TimerList:



Public Member Functions

- void Init ()

  Initialize the TimerList object.
- void Add (Timer *pclListNode_)

  Add a timer to the TimerList.
- void Remove (Timer *pclListNode_)

  Remove a timer from the TimerList, cancelling its expiry.
- void Process ()

  Process all timers in the timerlist as a result of the timer expiring.

Private Attributes

- K_ULONG m_ulNextWakeup

  The time (in system clock ticks) of the next wakeup event.

- K_UCHAR m_bTimerActive

  Whether or not the timer is active.

Additional Inherited Members

14.21.1 Detailed Description

TimerList class - a doubly-linked-list of timer objects.

Definition at line 261 of file timerlist.h.

14.21.2 Member Function Documentation

14.21.2.1 void TimerList::Add (Timer * *pclListNode_*)

Add a timer to the TimerList.

Parameters

*pclListNode_*	Pointer to the Timer to Add

Definition at line 49 of file timerlist.cpp.

14.21.2.2 void TimerList::Init (void)

Initialize the TimerList object.

Must be called before using the object.

Definition at line 42 of file timerlist.cpp.

14.21.2.3 void TimerList::Process (void)

Process all timers in the timerlist as a result of the timer expiring.

This will select a new timer epoch based on the next timer to expire. ToDo - figure out if we need to deal with any overtime here.

Definition at line 114 of file timerlist.cpp.

14.21.2.4 void TimerList::Remove (Timer * *pclListNode_*)

Remove a timer from the TimerList, cancelling its expiry.

Parameters

*pclListNode_*	Pointer to the Timer to remove

Definition at line 97 of file timerlist.cpp.

The documentation for this class was generated from the following files:

- timerlist.h
- timerlist.cpp

14.22 TimerScheduler Class Reference

"Static" Class used to interface a global TimerList with the rest of the kernel.

`#include <timerlist.h>`

Static Public Member Functions

- static void Init ()

  Initialize the timer scheduler.
- static void Add (Timer *pclListNode_)

  Add a timer to the timer scheduler.
- static void Remove (Timer *pclListNode_)

  Remove a timer from the timer scheduler.
- static void Process ()

  This function must be called on timer expiry (from the timer's ISR context).

Static Private Attributes

- static TimerList m_clTimerList

  TimerList object manipulated by the Timer Scheduler.

14.22.1 Detailed Description

"Static" Class used to interface a global TimerList with the rest of the kernel.

Definition at line 311 of file timerlist.h.

14.22.2 Member Function Documentation

14.22.2.1 void TimerScheduler::Add (Timer * *pclListNode_*) [inline], [static]

Add a timer to the timer scheduler.

Adding a timer implicitly starts the timer as well.

Parameters

*pclListNode_*	Pointer to the timer list node to add

Definition at line 330 of file timerlist.h.

14.22.2.2 void TimerScheduler::Init (void) [inline], [static]

Initialize the timer scheduler.

Must be called before any timer, or timer-derived functions are used.

Definition at line 320 of file timerlist.h.

14.22.2.3 void TimerScheduler::Process (void) [inline], [static]

This function must be called on timer expiry (from the timer's ISR context).

This will result in all timers being updated based on the epoch that just elapsed. New timer epochs are set based on the next timer to expire.

Definition at line 352 of file timerlist.h.

14.22.2.4 void TimerScheduler::Remove (Timer * *pclListNode_*) [inline],[static]

Remove a timer from the timer scheduler.

May implicitly stop the timer if this is the only active timer scheduled.

Parameters

*pclListNode_*	Pointer to the timer list node to remove

Definition at line 341 of file timerlist.h.

The documentation for this class was generated from the following files:

- timerlist.h
- timerlist.cpp

Chapter 15

File Documentation

15.1 atomic.cpp File Reference

Basic Atomic Operations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "atomic.h"
#include "threadport.h"
```

15.1.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.cpp.

15.2 atomic.cpp

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__   |__   ___| __   |__   ______
00004 |    \  /   | ||    \       ||      |     ||   |/ /     ||___    |
00005 |     \/    | ||     \      ||      \     ||     \      ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023 #include "atomic.h"
00024 #include "threadport.h"
00025
00026 #if KERNEL_USE_ATOMIC
00027
00028 //---------------------------------------------------------------------------
00029 K_UCHAR Atomic::Set( K_UCHAR *pucSource_, K_UCHAR ucVal_ )
00030 {
00031     K_UCHAR ucRet;
00032     CS_ENTER();
00033     ucRet = *pucSource_;
00034     *pucSource_ = ucVal_;
00035     CS_EXIT();
00036     return ucRet;
00037 }
00038 //---------------------------------------------------------------------------
00039 K_USHORT Atomic::Set( K_USHORT *pusSource_, K_USHORT usVal_ )
00040 {
```

```
00041     K_USHORT usRet;
00042     CS_ENTER();
00043     usRet = *pusSource_;
00044     *pusSource_ = usVal_;
00045     CS_EXIT();
00046     return usRet;
00047 }
00048 //---------------------------------------------------------------------------
00049 K_ULONG Atomic::Set( K_ULONG *pulSource_, K_ULONG ulVal_ )
00050 {
00051     K_ULONG ulRet;
00052     CS_ENTER();
00053     ulRet = *pulSource_;
00054     *pulSource_ = ulVal_;
00055     CS_EXIT();
00056     return ulRet;
00057 }
00058
00059 //---------------------------------------------------------------------------
00060 K_UCHAR Atomic::Add( K_UCHAR *pucSource_, K_UCHAR ucVal_ )
00061 {
00062     K_UCHAR ucRet;
00063     CS_ENTER();
00064     ucRet = *pucSource_;
00065     *pucSource_ += ucVal_;
00066     CS_EXIT();
00067     return ucRet;
00068 }
00069
00070 //---------------------------------------------------------------------------
00071 K_USHORT Atomic::Add( K_USHORT *pusSource_, K_USHORT usVal_ )
00072 {
00073     K_USHORT usRet;
00074     CS_ENTER();
00075     usRet = *pusSource_;
00076     *pusSource_ += usVal_;
00077     CS_EXIT();
00078     return usRet;
00079 }
00080
00081 //---------------------------------------------------------------------------
00082 K_ULONG Atomic::Add( K_ULONG *pulSource_, K_ULONG ulVal_ )
00083 {
00084     K_ULONG ulRet;
00085     CS_ENTER();
00086     ulRet = *pulSource_;
00087     *pulSource_ += ulVal_;
00088     CS_EXIT();
00089     return ulRet;
00090 }
00091
00092 //---------------------------------------------------------------------------
00093 K_UCHAR Atomic::Sub( K_UCHAR *pucSource_, K_UCHAR ucVal_ )
00094 {
00095     K_UCHAR ucRet;
00096     CS_ENTER();
00097     ucRet = *pucSource_;
00098     *pucSource_ -= ucVal_;
00099     CS_EXIT();
00100     return ucRet;
00101 }
00102
00103 //---------------------------------------------------------------------------
00104 K_USHORT Atomic::Sub( K_USHORT *pusSource_, K_USHORT usVal_ )
00105 {
00106     K_USHORT usRet;
00107     CS_ENTER();
00108     usRet = *pusSource_;
00109     *pusSource_ -= usVal_;
00110     CS_EXIT();
00111     return usRet;
00112 }
00113
00114 //---------------------------------------------------------------------------
00115 K_ULONG Atomic::Sub( K_ULONG *pulSource_, K_ULONG ulVal_ )
00116 {
00117     K_ULONG ulRet;
00118     CS_ENTER();
00119     ulRet = *pulSource_;
00120     *pulSource_ -= ulVal_;
00121     CS_EXIT();
00122     return ulRet;
00123 }
00124
00125 //---------------------------------------------------------------------------
00126 K_BOOL Atomic::TestAndSet( K_BOOL *pbLock_ )
00127 {
```

```
00128     K_UCHAR ucRet;
00129     CS_ENTER();
00130     ucRet = *pbLock_;
00131     if (!ucRet)
00132     {
00133         *pbLock_ = 1;
00134     }
00135     CS_EXIT();
00136     return ucRet;
00137 }
00138
00139 #endif // KERNEL_USE_ATOMIC
```

15.3 atomic.h File Reference

Basic Atomic Operations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "threadport.h"
```

15.3.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.h.

15.4 atomic.h

```
00001 /*============================================================================
00002       _____        _____        _____        _____
00003  ___|     |__  ___|     |__  ___|     |__  ___| __  |__  _______
00004 |     \   /  | ||     \       ||      |      ||  |/ /     ||___    |
00005 |      \/    | ||      \      ||      \      ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ============================================================================*/
00021 #ifndef __ATOMIC_H__
00022 #define __ATOMIC_H__
00023
00024 #include "kerneltypes.h"
00025 #include "mark3cfg.h"
00026 #include "threadport.h"
00027
00028 #if KERNEL_USE_ATOMIC
00029
00039 class Atomic
00040 {
00041 public:
00048     static K_UCHAR Set( K_UCHAR *pucSource_, K_UCHAR ucVal_ );
00049     static K_USHORT Set( K_USHORT *pusSource_, K_USHORT usVal_ );
00050     static K_ULONG Set( K_ULONG *pulSource_, K_ULONG ulVal_ );
00051
00058     static K_UCHAR Add( K_UCHAR *pucSource_, K_UCHAR ucVal_ );
00059     static K_USHORT Add( K_USHORT *pusSource_, K_USHORT usVal_ );
00060     static K_ULONG Add( K_ULONG *pulSource_, K_ULONG ulVal_ );
00061
00068     static K_UCHAR Sub( K_UCHAR *pucSource_, K_UCHAR ucVal_ );
00069     static K_USHORT Sub( K_USHORT *pusSource_, K_USHORT usVal_ );
00070     static K_ULONG Sub( K_ULONG *pulSource_, K_ULONG ulVal_ );
00071
00086     static K_BOOL TestAndSet( K_BOOL *pbLock );
00087 };
00088
00089 #endif // KERNEL_USE_ATOMIC
00090
00091 #endif //__ATOMIC_H__
```

15.5 blocking.cpp File Reference

Implementation of base class for blocking objects.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kernel_debug.h"
#include "blocking.h"
#include "thread.h"
```

Macros

- #define **__FILE_ID__** BLOCKING_CPP

15.5.1 Detailed Description

Implementation of base class for blocking objects.

Definition in file blocking.cpp.

15.6 blocking.cpp

```
00001 /*==========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ==========================================================================*/
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023 #include "kernel_debug.h"
00024
00025 #include "blocking.h"
00026 #include "thread.h"
00027
00028 //---------------------------------------------------------------------------
00029 #if defined __FILE_ID__
00030     #undef __FILE_ID__
00031 #endif
00032 #define __FILE_ID__     BLOCKING_CPP
00033
00034 #if KERNEL_USE_SEMAPHORE || KERNEL_USE_MUTEX
00035 //---------------------------------------------------------------------------
00036 void BlockingObject::Block(Thread *pclThread_)
00037 {
00038     KERNEL_ASSERT( pclThread_ );
00039     KERNEL_TRACE_1( STR_THREAD_BLOCK_1, (K_USHORT)pclThread_->GetID() );
00040
00041     // Remove the thread from its current thread list (the "owner" list)
00042     // ... And add the thread to this object's block list
00043     Scheduler::Remove(pclThread_);
00044     m_clBlockList.Add(pclThread_);
00045
00046     // Set the "current" list location to the blocklist for this thread
00047     pclThread_->SetCurrent(&m_clBlockList);
00048
00049 }
00050
00051 //---------------------------------------------------------------------------
00052 void BlockingObject::UnBlock(Thread *pclThread_)
00053 {
00054     KERNEL_ASSERT( pclThread_ );
00055     KERNEL_TRACE_1( STR_THREAD_UNBLOCK_1, (K_USHORT)pclThread_->GetID() );
00056
```

```
00057     // Remove the thread from its current thread list (the "owner" list)
00058     pclThread_->GetCurrent()->Remove(pclThread_);
00059 
00060     // Put the thread back in its active owner's list.  This is usually
00061     // the ready-queue at the thread's original priority.
00062     Scheduler::Add(pclThread_);
00063 
00064     // Tag the thread's current list location to its owner
00065     pclThread_->SetCurrent(pclThread_->GetOwner());
00066 }
00067 
00068 #endif
```

15.7 blocking.h File Reference

Blocking object base class declarations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
#include "threadlist.h"
#include "thread.h"
```

Classes

- class BlockingObject

  Class implementing thread-blocking primatives.

15.7.1 Detailed Description

Blocking object base class declarations. A Blocking object in Mark3 is essentially a thread list. Any blocking object implementation (being a semaphore, mutex, event flag, etc.) can be built on top of this class, utilizing the provided functions to manipulate thread location within the Kernel.

Blocking a thread results in that thread becoming de-scheduled, placed in the blocking object's own private list of threads which are waiting on the object.

Unblocking a thread results in the reverse: The thread is moved back to its original location from the blocking list.

The only difference between a blocking object based on this class is the logic used to determine what consitutes a Block or Unblock condition.

For instance, a semaphore Pend operation may result in a call to the Block() method with the currently-executing thread in order to make that thread wait for a semaphore Post. That operation would then invoke the UnBlock() method, removing the blocking thread from the semaphore's list, and back into the the appropriate thread inside the scheduler.

Care must be taken when implementing blocking objects to ensure that critical sections are used judiciously, otherwise asynchronous events like timers and interrupts could result in non-deterministic and often catastrophic behavior.

Definition in file blocking.h.

15.8 blocking.h

```
00001 /*===========================================================================
00002     _____       _____       _____       _____
00003  ___|    _|__  __|_    |__  __|__    |__  __| __  __|__  _______
00004 |    \  /   | ||    \      ||     |     ||  |/ /     ||___    |
00005 |     \/    | ||     \     ||     \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |_____|       |_____|       |_____|       |_____|
```

```
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00047 #ifndef __BLOCKING_H__
00048 #define __BLOCKING_H__
00049
00050 #include "kerneltypes.h"
00051 #include "mark3cfg.h"
00052
00053 #include "ll.h"
00054 #include "threadlist.h"
00055 #include "thread.h"
00056
00057 #if KERNEL_USE_MUTEX || KERNEL_USE_SEMAPHORE || KERNEL_USE_EVENTFLAG
00058
00059 //---------------------------------------------------------------------------
00065 class BlockingObject
00066 {
00067 protected:
00088     void Block(Thread *pclThread_ );
00089
00101     void UnBlock(Thread *pclThread_);
00102
00107     ThreadList m_clBlockList;
00108 };
00109
00110 #endif
00111
00112 #endif
```

15.9 debug_tokens.h File Reference

Hex codes/translation tables used for efficient string tokenization.

Macros

- #define BLOCKING_CPP 0x0001 /* SUBSTITUTE="blocking.cpp" */

  Source file names start at 0x0000.
- #define **DRIVER_CPP** 0x0002 /* SUBSTITUTE="driver.cpp" */
- #define **KERNEL_CPP** 0x0003 /* SUBSTITUTE="kernel.cpp" */
- #define **LL_CPP** 0x0004 /* SUBSTITUTE="ll.cpp" */
- #define **MESSAGE_CPP** 0x0005 /* SUBSTITUTE="message.cpp" */
- #define **MUTEX_CPP** 0x0006 /* SUBSTITUTE="mutex.cpp" */
- #define **PROFILE_CPP** 0x0007 /* SUBSTITUTE="profile.cpp" */
- #define **QUANTUM_CPP** 0x0008 /* SUBSTITUTE="quantum.cpp" */
- #define **SCHEDULER_CPP** 0x0009 /* SUBSTITUTE="scheduler.cpp" */
- #define **SEMAPHORE_CPP** 0x000A /* SUBSTITUTE="semaphore.cpp" */
- #define **THREAD_CPP** 0x000B /* SUBSTITUTE="thread.cpp" */
- #define **THREADLIST_CPP** 0x000C /* SUBSTITUTE="threadlist.cpp" */
- #define **TIMERLIST_CPP** 0x000D /* SUBSTITUTE="timerlist.cpp" */
- #define **KERNELSWI_CPP** 0x000E /* SUBSTITUTE="kernelswi.cpp" */
- #define **KERNELTIMER_CPP** 0x000F /* SUBSTITUTE="kerneltimer.cpp" */
- #define **KPROFILE_CPP** 0x0010 /* SUBSTITUTE="kprofile.cpp" */
- #define **THREADPORT_CPP** 0x0011 /* SUBSTITUTE="threadport.cpp" */
- #define BLOCKING_H 0x1000 /* SUBSTITUTE="blocking.h" */

  Header file names start at 0x1000.
- #define **DRIVER_H** 0x1001 /* SUBSTITUTE="driver.h" */
- #define **KERNEL_H** 0x1002 /* SUBSTITUTE="kernel.h" */
- #define **KERNELTYPES_H** 0x1003 /* SUBSTITUTE="kerneltypes.h" */
- #define **LL_H** 0x1004 /* SUBSTITUTE="ll.h" */
- #define **MANUAL_H** 0x1005 /* SUBSTITUTE="manual.h" */

- #define **MARK3CFG_H** 0x1006 /* SUBSTITUTE="mark3cfg.h" */
- #define **MESSAGE_H** 0x1007 /* SUBSTITUTE="message.h" */
- #define **MUTEX_H** 0x1008 /* SUBSTITUTE="mutex.h" */
- #define **PROFILE_H** 0x1009 /* SUBSTITUTE="profile.h" */
- #define **PROFILING_RESULTS_H** 0x100A /* SUBSTITUTE="profiling_results.h" */
- #define **QUANTUM_H** 0x100B /* SUBSTITUTE="quantum.h" */
- #define **SCHEDULER_H** 0x100C /* SUBSTITUTE="scheduler.h" */
- #define **SEMAPHORE_H** 0x100D /* SUBSTITUTE="ksemaphore.h" */
- #define **THREAD_H** 0x100E /* SUBSTITUTE="thread.h" */
- #define **THREADLIST_H** 0x100F /* SUBSTITUTE="threadlist.h" */
- #define **TIMERLIST_H** 0x1010 /* SUBSTITUTE="timerlist.h" */
- #define **KERNELSWI_H** 0x1011 /* SUBSTITUTE="kernelswi.h */
- #define **KERNELTIMER_H** 0x1012 /* SUBSTITUTE="kerneltimer.h */
- #define **KPROFILE_H** 0x1013 /* SUBSTITUTE="kprofile.h" */
- #define **THREADPORT_H** 0x1014 /* SUBSTITUTE="threadport.h" */
- #define STR_PANIC 0x2000 /* SUBSTITUTE="!Panic!" */

  Indexed strings start at 0x2000.
- #define **STR_MARK3_INIT** 0x2001 /* SUBSTITUTE="Initializing Kernel Objects" */
- #define **STR_KERNEL_ENTER** 0x2002 /* SUBSTITUTE="Starting Kernel" */
- #define **STR_THREAD_START** 0x2003 /* SUBSTITUTE="Switching to First Thread" */
- #define **STR_START_ERROR** 0x2004 /* SUBSTITUTE="Error starting kernel - function should never return" */
- #define **STR_THREAD_CREATE** 0x2005 /* SUBSTITUTE="Creating Thread" */
- #define **STR_STACK_SIZE_1** 0x2006 /* SUBSTITUTE=" Stack Size: %1" */
- #define **STR_PRIORITY_1** 0x2007 /* SUBSTITUTE=" Priority: %1" */
- #define **STR_THREAD_ID_1** 0x2008 /* SUBSTITUTE=" Thread ID: %1" */
- #define **STR_ENTRYPOINT_1** 0x2009 /* SUBSTITUTE=" EntryPoint: %1" */
- #define **STR_CONTEXT_SWITCH_1** 0x200A /* SUBSTITUTE="Context Switch To Thread: %1" */
- #define **STR_IDLING** 0x200B /* SUBSTITUTE="Idling CPU" */
- #define **STR_WAKEUP** 0x200C /* SUBSTITUTE="Waking up" */
- #define **STR_SEMAPHORE_PEND_1** 0x200D /* SUBSTITUTE="Semaphore Pend: %1" */
- #define **STR_SEMAPHORE_POST_1** 0x200E /* SUBSTITUTE="Semaphore Post: %1" */
- #define **STR_MUTEX_CLAIM_1** 0x200F /* SUBSTITUTE="Mutex Claim: %1" */
- #define **STR_MUTEX_RELEASE_1** 0x2010 /* SUBSTITUTE="Mutex Release: %1" */
- #define **STR_THREAD_BLOCK_1** 0x2011 /* SUBSTITUTE="Thread %1 Blocked" */
- #define **STR_THREAD_UNBLOCK_1** 0x2012-2015 /* SUBSTITUTE="Thread %1 Unblocked" */
- #define **STR_ASSERT_FAILED** 0x2013 /* SUBSTITUTE="Assertion Failed" */
- #define **STR_SCHEDULE_1** 0x2014 /* SUBSTITUTE="Scheduler chose %1" */
- #define **STR_THREAD_START_1** 0x2015 /* SUBSTITUTE="Thread Start: %1" */
- #define **STR_THREAD_EXIT_1** 0x2016 /* SUBSTITUTE="Thread Exit: %1" */
- #define **STR_UNDEFINED** 0xFFFF /* SUBSTITUTE="UNDEFINED" */

15.9.1 Detailed Description

Hex codes/translation tables used for efficient string tokenization. We use this for efficiently encoding strings used for kernel traces, debug prints, etc. The upside - this is really fast and efficient for encoding strings and data. Downside? The tools need to parse this header file in order to convert the enumerated data into actual strings, decoding them.

Definition in file debug_tokens.h.

15.10 debug_tokens.h

```
00001 /*=============================================================================
00002     ______     ______     ______     __  __
00003    /\  ___\   /\  ___\   /\  ___\   /\ \/ /
00004    \ \___  \  \ \ \____  \ \ \____  \ \  _"-.
00005     \/\_____\  \ \_____\  \ \_____\  \ \_\ \_\
00006      \/_____/   \/_____/   \/_____/   \/_/\/_/
00007
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =============================================================================*/
00025 #ifndef __DEBUG_TOKENS_H__
00026 #define __DEBUG_TOKENS_H__
00027 //---------------------------------------------------------------------------
00029 #define BLOCKING_CPP         0x0001      /* SUBSTITUTE="blocking.cpp" */
00030 #define DRIVER_CPP           0x0002      /* SUBSTITUTE="driver.cpp" */
00031 #define KERNEL_CPP           0x0003      /* SUBSTITUTE="kernel.cpp" */
00032 #define LL_CPP               0x0004      /* SUBSTITUTE="ll.cpp" */
00033 #define MESSAGE_CPP          0x0005      /* SUBSTITUTE="message.cpp" */
00034 #define MUTEX_CPP            0x0006      /* SUBSTITUTE="mutex.cpp" */
00035 #define PROFILE_CPP          0x0007      /* SUBSTITUTE="profile.cpp" */
00036 #define QUANTUM_CPP          0x0008      /* SUBSTITUTE="quantum.cpp" */
00037 #define SCHEDULER_CPP        0x0009      /* SUBSTITUTE="scheduler.cpp" */
00038 #define SEMAPHORE_CPP        0x000A      /* SUBSTITUTE="semaphore.cpp" */
00039 #define THREAD_CPP           0x000B      /* SUBSTITUTE="thread.cpp" */
00040 #define THREADLIST_CPP       0x000C      /* SUBSTITUTE="threadlist.cpp" */
00041 #define TIMERLIST_CPP        0x000D      /* SUBSTITUTE="timerlist.cpp" */
00042 #define KERNELSWI_CPP        0x000E      /* SUBSTITUTE="kernelswi.cpp" */
00043 #define KERNELTIMER_CPP      0x000F      /* SUBSTITUTE="kerneltimer.cpp" */
00044 #define KPROFILE_CPP         0x0010      /* SUBSTITUTE="kprofile.cpp" */
00045 #define THREADPORT_CPP       0x0011      /* SUBSTITUTE="threadport.cpp" */
00046
00047 //---------------------------------------------------------------------------
00049 #define BLOCKING_H           0x1000      /* SUBSTITUTE="blocking.h" */
00050 #define DRIVER_H             0x1001      /* SUBSTITUTE="driver.h" */
00051 #define KERNEL_H             0x1002      /* SUBSTITUTE="kernel.h" */
00052 #define KERNELTYPES_H        0x1003      /* SUBSTITUTE="kerneltypes.h" */
00053 #define LL_H                 0x1004      /* SUBSTITUTE="ll.h" */
00054 #define MANUAL_H             0x1005      /* SUBSTITUTE="manual.h" */
00055 #define MARK3CFG_H           0x1006      /* SUBSTITUTE="mark3cfg.h" */
00056 #define MESSAGE_H            0x1007      /* SUBSTITUTE="message.h" */
00057 #define MUTEX_H              0x1008      /* SUBSTITUTE="mutex.h" */
00058 #define PROFILE_H            0x1009      /* SUBSTITUTE="profile.h" */
00059 #define PROFILING_RESULTS_H  0x100A      /* SUBSTITUTE="profiling_results.h" */
00060 #define QUANTUM_H            0x100B      /* SUBSTITUTE="quantum.h" */
00061 #define SCHEDULER_H          0x100C      /* SUBSTITUTE="scheduler.h" */
00062 #define SEMAPHORE_H          0x100D      /* SUBSTITUTE="ksemaphore.h" */
00063 #define THREAD_H             0x100E      /* SUBSTITUTE="thread.h" */
00064 #define THREADLIST_H         0x100F      /* SUBSTITUTE="threadlist.h" */
00065 #define TIMERLIST_H          0x1010      /* SUBSTITUTE="timerlist.h" */
00066 #define KERNELSWI_H          0x1011      /* SUBSTITUTE="kernelswi.h */
00067 #define KERNELTIMER_H        0x1012      /* SUBSTITUTE="kerneltimer.h */
00068 #define KPROFILE_H           0x1013      /* SUBSTITUTE="kprofile.h" */
00069 #define THREADPORT_H         0x1014      /* SUBSTITUTE="threadport.h" */
00070
00071 //---------------------------------------------------------------------------
00073 #define STR_PANIC                0x2000      /* SUBSTITUTE="!Panic!" */
00074 #define STR_MARK3_INIT           0x2001      /* SUBSTITUTE="Initializing Kernel Objects" */
00075 #define STR_KERNEL_ENTER         0x2002      /* SUBSTITUTE="Starting Kernel" */
00076 #define STR_THREAD_START         0x2003      /* SUBSTITUTE="Switching to First Thread" */
00077 #define STR_START_ERROR          0x2004      /* SUBSTITUTE="Error starting kernel - function should never
      return" */
00078 #define STR_THREAD_CREATE        0x2005      /* SUBSTITUTE="Creating Thread" */
00079 #define STR_STACK_SIZE_1         0x2006      /* SUBSTITUTE="  Stack Size: %1" */
00080 #define STR_PRIORITY_1           0x2007      /* SUBSTITUTE="  Priority: %1" */
00081 #define STR_THREAD_ID_1          0x2008      /* SUBSTITUTE="  Thread ID: %1" */
00082 #define STR_ENTRYPOINT_1         0x2009      /* SUBSTITUTE="  EntryPoint: %1" */
00083 #define STR_CONTEXT_SWITCH_1     0x200A      /* SUBSTITUTE="Context Switch To Thread: %1" */
00084 #define STR_IDLING               0x200B      /* SUBSTITUTE="Idling CPU" */
00085 #define STR_WAKEUP               0x200C      /* SUBSTITUTE="Waking up" */
00086 #define STR_SEMAPHORE_PEND_1     0x200D      /* SUBSTITUTE="Semaphore Pend: %1" */
00087 #define STR_SEMAPHORE_POST_1     0x200E      /* SUBSTITUTE="Semaphore Post: %1" */
00088 #define STR_MUTEX_CLAIM_1        0x200F      /* SUBSTITUTE="Mutex Claim: %1" */
00089 #define STR_MUTEX_RELEASE_1      0x2010      /* SUBSTITUTE="Mutex Release: %1" */
00090 #define STR_THREAD_BLOCK_1       0x2011      /* SUBSTITUTE="Thread %1 Blocked" */
00091 #define STR_THREAD_UNBLOCK_1     0x2012-2015     /* SUBSTITUTE="Thread %1 Unblocked" */
00092 #define STR_ASSERT_FAILED        0x2013      /* SUBSTITUTE="Assertion Failed" */
00093 #define STR_SCHEDULE_1           0x2014      /* SUBSTITUTE="Scheduler chose %1" */
00094 #define STR_THREAD_START_1       0x2015      /* SUBSTITUTE="Thread Start: %1" */
00095 #define STR_THREAD_EXIT_1        0x2016      /* SUBSTITUTE="Thread Exit: %1" */
00096
00097 //---------------------------------------------------------------------------
```

```
00098 #define STR_UNDEFINED           0xFFFF      /* SUBSTITUTE="UNDEFINED" */
00099 #endif
```

15.11 driver.cpp File Reference

Device driver/hardware abstraction layer.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kernel_debug.h"
#include "driver.h"
```

Macros

- #define **__FILE_ID__** DRIVER_CPP

15.11.1 Detailed Description

Device driver/hardware abstraction layer.

Definition in file driver.cpp.

15.12 driver.cpp

```
00001 /*===========================================================================
00002       _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \   /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/    | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023 #include "kernel_debug.h"
00024 #include "driver.h"
00025
00026 //---------------------------------------------------------------------------
00027 #if defined __FILE_ID__
00028     #undef __FILE_ID__
00029 #endif
00030 #define __FILE_ID__     DRIVER_CPP
00031
00032 //---------------------------------------------------------------------------
00033 #if KERNEL_USE_DRIVER
00034
00035 DoubleLinkList DriverList::m_clDriverList;
00036
00040 class DevNull : public Driver
00041 {
00042 public:
00043     virtual void Init() { SetName("/dev/null"); };
00044     virtual K_UCHAR Open() { return 0; }
00045     virtual K_UCHAR Close() { return 0; }
00046
00047     virtual K_USHORT Read( K_USHORT usBytes_,
00048     K_UCHAR *pucData_){ return 0; }
00049
00050     virtual K_USHORT Write( K_USHORT usBytes_,
00051     K_UCHAR *pucData_) { return 0; }
00052
00053     virtual K_USHORT Control( K_USHORT usEvent_,
00054         void *pvDataIn_,
```

```
00055         K_USHORT usSizeIn_,
00056         void *pvDataOut_,
00057         K_USHORT usSizeOut_ ) { return 0; }
00058
00059 };
00060
00061 //---------------------------------------------------------------------------
00062 static DevNull clDevNull;
00063
00064 //---------------------------------------------------------------------------
00065 static K_UCHAR DrvCmp( const K_CHAR *szStr1_, const K_CHAR *szStr2_ )
00066 {
00067     K_CHAR *szTmp1 = (K_CHAR*) szStr1_;
00068     K_CHAR *szTmp2 = (K_CHAR*) szStr2_;
00069
00070     while (*szTmp1 && *szTmp2)
00071     {
00072         if (*szTmp1++ != *szTmp2++)
00073         {
00074             return 0;
00075         }
00076     }
00077
00078     // Both terminate at the same length
00079     if (!(*szTmp1) && !(*szTmp2))
00080     {
00081         return 1;
00082     }
00083
00084     return 0;
00085 }
00086
00087 //---------------------------------------------------------------------------
00088 void DriverList::Init()
00089 {
00090     // Ensure we always have at least one entry - a default in case no match
00091     // is found (/dev/null)
00092     clDevNull.Init();
00093     Add(&clDevNull);
00094 }
00095
00096 //---------------------------------------------------------------------------
00097 Driver *DriverList::FindByPath( const K_CHAR *m_pcPath )
00098 {
00099     KERNEL_ASSERT( m_pcPath );
00100     Driver *pclTemp = static_cast<Driver*>(m_clDriverList.GetHead());
00101
00102     while (pclTemp)
00103     {
00104         if(DrvCmp(m_pcPath, pclTemp->GetPath()))
00105         {
00106             return pclTemp;
00107         }
00108         pclTemp = static_cast<Driver*>(pclTemp->GetNext());
00109     }
00110     return &clDevNull;
00111 }
00112
00113 #endif
```

15.13 driver.h File Reference

Driver abstraction framework.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
```

15.13.1 Detailed Description

Driver abstraction framework.

15.13.2 Intro

This is the basis of the driver framework. In the context of Mark3, drivers don't necessarily have to be based on physical hardware peripherals. They can be used to represent algorithms (such as random number generators), files, or protocol stacks. Unlike FunkOS, where driver IO is protected automatically by a mutex, we do not use this kind of protection - we leave it up to the driver implementor to do what's right in its own context. This also frees up the driver to implement all sorts of other neat stuff, like sending messages to threads associated with the driver. Drivers are implemented as character devices, with the standard array of posix-style accessor methods for reading, writing, and general driver control.

A global driver list is provided as a convenient and minimal "filesystem" structure, in which devices can be accessed by name.

15.13.3 Driver Design

A device driver needs to be able to perform the following operations: -Initialize a peripheral -Start/stop a peripheral -Handle I/O control operations -Perform various read/write operations

At the end of the day, that's pretty much all a device driver has to do, and all of the functionality that needs to be presented to the developer.

We abstract all device drivers using a base-class which implements the following methods: -Start/Open -Stop/Close -Control -Read -Write

A basic driver framework and API can thus be implemented in five function calls - that's it! You could even reduce that further by handling the initialize, start, and stop operations inside the "control" operation.

15.13.4 Driver API

In C++, we can implement this as a class to abstract these event handlers, with virtual void functions in the base class overridden by the inherited objects.

To add and remove device drivers from the global table, we use the following methods:

```
void DriverList::Add( Driver *pclDriver_ );
void DriverList::Remove( Driver *pclDriver_ );
```

DriverList::Add()/Remove() takes a single arguments the pointer to he object to operate on.

Once a driver has been added to the table, drivers are opened by NAME using DriverList::FindByName("/dev/name"). This function returns a pointer to the specified driver if successful, or to a built in /dev/null device if the path name is invalid. After a driver is open, that pointer is used for all other driver access functions.

This abstraction is incredibly useful any peripheral or service can be accessed through a consistent set of APIs, that make it easy to substitute implementations from one platform to another. Portability is ensured, the overhead is negligible, and it emphasizes the reuse of both driver and application code as separate entities.

Consider a system with drivers for I2C, SPI, and UART peripherals - under our driver framework, an application can initialize these peripherals and write a greeting to each using the same simple API functions for all drivers:

```
pclI2C  = DriverList::FindByName("/dev/i2c");
pclUART = DriverList::FindByName("/dev/tty0");
pclSPI  = DriverList::FindByName("/dev/spi");

pclI2C->Write(12,"Hello World!");
pclUART->Write(12, "Hello World!");
pclSPI->Write(12, "Hello World!");
```

Definition in file driver.h.

15.14 driver.h

```
00001 /*=========================================================================
```

```
00002       _____          _____          _____          _____
00003  ____|    _|__   ___|_    |__   ___|__   |__   ___| __  |__   ______
00004 |    \   /   | ||     \       ||       |      ||   |/ /      ||___    |
00005 |     \/    | ||      \      ||       \      ||      \      ||___    |
00006 |__/\__/|__|_||__|\__\   __||__|\__\   __||__|\__\   __||______|
00007       |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ============================================================================*/
00105 #include "kerneltypes.h"
00106 #include "mark3cfg.h"
00107
00108 #include "ll.h"
00109
00110 #ifndef __DRIVER_H__
00111 #define __DRIVER_H__
00112
00113 #if KERNEL_USE_DRIVER
00114
00115 class DriverList;
00116 //---------------------------------------------------------------------------
00121 class Driver : public LinkListNode
00122 {
00123 public:
00129     virtual void Init() = 0;
00130
00138     virtual K_UCHAR Open() = 0;
00139
00147     virtual K_UCHAR Close() = 0;
00148
00164     virtual K_USHORT Read( K_USHORT usBytes_,
00165                            K_UCHAR *pucData_) = 0;
00166
00183     virtual K_USHORT Write( K_USHORT usBytes_,
00184                             K_UCHAR *pucData_) = 0;
00185
00208     virtual K_USHORT Control( K_USHORT usEvent_,
00209                               void *pvDataIn_,
00210                               K_USHORT usSizeIn_,
00211                               void *pvDataOut_,
00212                               K_USHORT usSizeOut_ ) = 0;
00213
00222     void SetName( const K_CHAR *pcName_ ) { m_pcPath = pcName_; }
00223
00231     const K_CHAR *GetPath() { return m_pcPath; }
00232
00233 private:
00234
00236     const K_CHAR *m_pcPath;
00237 };
00238
00239 //---------------------------------------------------------------------------
00244 class DriverList
00245 {
00246 public:
00254     static void Init();
00255
00264     static void Add( Driver *pclDriver_ ) { m_clDriverList.Add(pclDriver_); }
00265
00274     static void Remove( Driver *pclDriver_ ) { m_clDriverList.Remove(pclDriver_); }
00275
00282     static Driver *FindByPath( const K_CHAR *m_pcPath );
00283
00284 private:
00285
00287     static DoubleLinkList m_clDriverList;
00288 };
00289
00290 #endif //KERNEL_USE_DRIVER
00291
00292 #endif
```

15.15 eventflag.cpp File Reference

Event Flag Blocking Object/IPC-Object implementation.

```
#include "mark3cfg.h"
#include "blocking.h"
#include "kernel.h"
#include "thread.h"
#include "eventflag.h"
```

15.15.1 Detailed Description

Event Flag Blocking Object/IPC-Object implementation.

Definition in file eventflag.cpp.

15.16 eventflag.cpp

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|     |__   ___|     |___  ___|     |___   ___| ___  |___  ______
00004 |    \   /  | ||    \       ||     |      ||   |/ /     ||___    |
00005 |     \/    | ||     \      ||     \      ||     \      ||___    |
00006 |__/\__/|__|_||__|\__\  ___||__|\__\  ___||__|\__\  ___||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00019 #include "mark3cfg.h"
00020 #include "blocking.h"
00021 #include "kernel.h"
00022 #include "thread.h"
00023 #include "eventflag.h"
00024
00025 #if KERNEL_USE_EVENTFLAG
00026
00027 #if KERNEL_USE_TIMEOUTS
00028 #include "timerlist.h"
00029 //---------------------------------------------------------------------------
00030 void TimedEventFlag_Callback(Thread *pclOwner_, void *pvData_)
00031 {
00032     EventFlag *pclEventFlag = static_cast<EventFlag*>(pvData_);
00033
00034     pclEventFlag->WakeMe(pclOwner_);
00035     pclOwner_->SetExpired(true);
00036     pclOwner_->SetEventFlagMask(0);
00037
00038     if (pclOwner_->GetPriority() > Scheduler::GetCurrentThread()->
      GetPriority())
00039     {
00040         Thread::Yield();
00041     }
00042 }
00043
00044 //---------------------------------------------------------------------------
00045 void EventFlag::WakeMe(Thread *pclChosenOne_)
00046 {
00047     UnBlock(pclChosenOne_);
00048 }
00049 #endif
00050
00051 //---------------------------------------------------------------------------
00052 #if KERNEL_USE_TIMEOUTS
00053     K_USHORT EventFlag::Wait_i(K_USHORT usMask_, EventFlagOperation_t eMode_, K_ULONG
      ulTimeMS_)
00054 #else
00055     K_USHORT EventFlag::Wait_i(K_USHORT usMask_, EventFlagOperation_t eMode_)
00056 #endif
00057 {
00058     bool bThreadYield = false;
00059     bool bMatch = false;
00060
00061 #if KERNEL_USE_TIMEOUTS
00062     Timer clEventTimer;
00063     bool bUseTimer = false;
00064 #endif
00065
```

```
00066     // Ensure we're operating in a critical section while we determine
00067     // whether or not we need to block the current thread on this object.
00068     CS_ENTER();
00069 
00070     // Check to see whether or not the current mask matches any of the
00071     // desired bits.
00072     g_pstCurrent->SetEventFlagMask(usMask_);
00073 
00074     if ((eMode_ == EVENT_FLAG_ALL) || (eMode_ == EVENT_FLAG_ALL_CLEAR))
00075     {
00076         // Check to see if the flags in their current state match all of
00077         // the set flags in the event flag group, with this mask.
00078         if ((m_usSetMask & usMask_) == usMask_)
00079         {
00080             bMatch = true;
00081             g_pstCurrent->SetEventFlagMask(usMask_);
00082         }
00083     }
00084     else if ((eMode_ == EVENT_FLAG_ANY) || (eMode_ == EVENT_FLAG_ANY_CLEAR))
00085     {
00086         // Check to see if the existing flags match any of the set flags in
00087         // the event flag group  with this mask
00088         if (m_usSetMask & usMask_)
00089         {
00090             bMatch = true;
00091             g_pstCurrent->SetEventFlagMask(m_usSetMask & usMask_);
00092         }
00093     }
00094 
00095     // We're unable to match this pattern as-is, so we must block.
00096     if (!bMatch)
00097     {
00098         // Reset the current thread's event flag mask & mode
00099         g_pstCurrent->SetEventFlagMask(usMask_);
00100         g_pstCurrent->SetEventFlagMode(eMode_);
00101 
00102 #if KERNEL_USE_TIMEOUTS
00103         if (ulTimeMS_)
00104         {
00105             g_pstCurrent->SetExpired(false);
00106             clEventTimer.Init();
00107             clEventTimer.Start(0, ulTimeMS_, TimedEventFlag_Callback, (void*)this);
00108             bUseTimer = true;
00109         }
00110 #endif
00111 
00112         // Add the thread to the object's block-list.
00113         Block(g_pstCurrent);
00114 
00115         // Trigger that
00116         bThreadYield = true;
00117     }
00118 
00119     // If bThreadYield is set, it means that we've blocked the current thread,
00120     // and must therefore rerun the scheduler to determine what thread to
00121     // switch to.
00122     if (bThreadYield)
00123     {
00124         // Switch threads immediately
00125         Thread::Yield();
00126     }
00127 
00128     // Exit the critical section and return back to normal execution
00129     CS_EXIT();
00130 
00135 #if KERNEL_USE_TIMEOUTS
00136     if (bUseTimer && bThreadYield)
00137     {
00138         clEventTimer.Stop();
00139     }
00140 #endif
00141 
00142     return g_pstCurrent->GetEventFlagMask();
00143 }
00144 
00145 //---------------------------------------------------------------------------
00146 K_USHORT EventFlag::Wait(K_USHORT usMask_, EventFlagOperation_t eMode_)
00147 {
00148 #if KERNEL_USE_TIMEOUTS
00149     return Wait_i(usMask_, eMode_, 0);
00150 #else
00151     return Wait_i(usMask_, eMode_);
00152 #endif
00153 }
00154 
00155 #if KERNEL_USE_TIMEOUTS
00156 //---------------------------------------------------------------------------
```

```
00157 K_USHORT EventFlag::Wait(K_USHORT usMask_, EventFlagOperation_t eMode_, K_ULONG ulTimeMS_)
00158 {
00159     return Wait_i(usMask_, eMode_, ulTimeMS_);
00160 }
00161 #endif
00162
00163 //---------------------------------------------------------------------------
00164 void EventFlag::Set(K_USHORT usMask_)
00165 {
00166     Thread *pclPrev;
00167     Thread *pclCurrent;
00168     bool bReschedule = false;
00169     K_USHORT usNewMask;
00170
00171     CS_ENTER();
00172
00173     // Walk through the whole block list, checking to see whether or not
00174     // the current flag set now matches any/all of the masks and modes of
00175     // the threads involved.
00176
00177     m_usSetMask |= usMask_;
00178     usNewMask = m_usSetMask;
00179
00180     // Start at the head of the list, and iterate through until we hit the
00181     // "head" element in the list again.  Ensure that we handle the case where
00182     // we remove the first or last elements in the list, or if there's only
00183     // one element in the list.
00184     pclCurrent = static_cast<Thread*>(m_clBlockList.GetHead());
00185
00186     // Do nothing when there are no objects blocking.
00187     if (pclCurrent)
00188     {
00189         // First loop - process every thread in the block-list and check to
00190         // see whether or not the current flags match the event-flag conditions
00191         // on the thread.
00192         do
00193         {
00194             pclPrev = pclCurrent;
00195             pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00196
00197             // Read the thread's event mask/mode
00198             K_USHORT usThreadMask = pclPrev->GetEventFlagMask();
00199             EventFlagOperation_t eThreadMode = pclPrev->GetEventFlagMode();
00200
00201             // For the "any" mode - unblock the blocked threads if one or more bits
00202             // in the thread's bitmask match the object's bitmask
00203             if ((EVENT_FLAG_ANY == eThreadMode) || (EVENT_FLAG_ANY_CLEAR == eThreadMode))
00204             {
00205                 if (usThreadMask & m_usSetMask)
00206                 {
00207                     pclPrev->SetEventFlagMode(EVENT_FLAG_PENDING_UNBLOCK);
00208                     pclPrev->SetEventFlagMask(m_usSetMask & usThreadMask);
00209                     bReschedule = true;
00210
00211                     // If the "clear" variant is set, then clear the bits in the mask
00212                     // that caused the thread to unblock.
00213                     if (EVENT_FLAG_ANY_CLEAR == eThreadMode)
00214                     {
00215                         usNewMask &=~ (usThreadMask & usMask_);
00216                     }
00217                 }
00218             }
00219             // For the "all" mode, every set bit in the thread's requested bitmask must
00220             // match the object's flag mask.
00221             else if ((EVENT_FLAG_ALL == eThreadMode) || (EVENT_FLAG_ALL_CLEAR == eThreadMode))
00222             {
00223                 if ((usThreadMask & m_usSetMask) == usThreadMask)
00224                 {
00225                     pclPrev->SetEventFlagMode(EVENT_FLAG_PENDING_UNBLOCK);
00226                     pclPrev->SetEventFlagMask(usThreadMask);
00227                     bReschedule = true;
00228
00229                     // If the "clear" variant is set, then clear the bits in the mask
00230                     // that caused the thread to unblock.
00231                     if (EVENT_FLAG_ALL_CLEAR == eThreadMode)
00232                     {
00233                         usNewMask &=~ (usThreadMask & usMask_);
00234                     }
00235                 }
00236             }
00237         }
00238         // To keep looping, ensure that there's something in the list, and
00239         // that the next item isn't the head of the list.
00240         while (pclPrev != m_clBlockList.GetTail());
00241
00242         // Second loop - go through and unblock all of the threads that
00243         // were tagged for unblocking.
```

```
00244         pclCurrent = static_cast<Thread*>(m_clBlockList.
      GetHead());
00245         bool bIsTail = false;
00246         do
00247         {
00248             pclPrev = pclCurrent;
00249             pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00250 
00251             // Check to see if this is the condition to terminate the loop
00252             if (pclPrev == m_clBlockList.GetTail())
00253             {
00254                 bIsTail = true;
00255             }
00256 
00257             // If the first pass indicated that this thread should be
00258             // unblocked, then unblock the thread
00259             if (pclPrev->GetEventFlagMode() == EVENT_FLAG_PENDING_UNBLOCK)
00260             {
00261                 UnBlock(pclPrev);
00262             }
00263         }
00264         while (!bIsTail);
00265     }
00266 
00267     // If we awoke any threads, re-run the scheduler
00268     if (bReschedule)
00269     {
00270         Thread::Yield();
00271     }
00272 
00273     // Update the bitmask based on any "clear" operations performed along
00274     // the way
00275     m_usSetMask = usNewMask;
00276 
00277     // Restore interrupts - will potentially cause a context switch if a
00278     // thread is unblocked.
00279     CS_EXIT();
00280 }
00281 
00282 //---------------------------------------------------------------------------
00283 void EventFlag::Clear(K_USHORT usMask_)
00284 {
00285     // Just clear the bitfields in the local object.
00286     CS_ENTER();
00287     m_usSetMask &= ~usMask_;
00288     CS_EXIT();
00289 }
00290 
00291 //---------------------------------------------------------------------------
00292 K_USHORT EventFlag::GetMask()
00293 {
00294     // Return the presently held event flag values in this object.  Ensure
00295     // we get this within a critical section to guarantee atomicity.
00296     K_USHORT usReturn;
00297     CS_ENTER();
00298     usReturn = m_usSetMask;
00299     CS_EXIT();
00300     return usReturn;
00301 }
00302 
00303 #endif // KERNEL_USE_EVENTFLAG
```

15.17 eventflag.h File Reference

Event Flag Blocking Object/IPC-Object definition.

```
#include "mark3cfg.h"
#include "kernel.h"
#include "kerneltypes.h"
#include "blocking.h"
#include "thread.h"
```

Classes

- class EventFlag

The EventFlag class is a blocking object, similar to a semaphore or mutex, commonly used for synchronizing thread execution based on events occurring within the system.

15.17.1 Detailed Description

Event Flag Blocking Object/IPC-Object definition.

Definition in file eventflag.h.

15.18 eventflag.h

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__   |__   ___| __   |__   ______
00004 |    \  /   | ||     \       ||      |      ||   |/ /      ||___    |
00005 |     \/    | ||      \      ||      \      ||     \      ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00019 #ifndef __EVENTFLAG_H__
00020 #define __EVENTFLAG_H__
00021
00022 #include "mark3cfg.h"
00023 #include "kernel.h"
00024 #include "kerneltypes.h"
00025 #include "blocking.h"
00026 #include "thread.h"
00027
00028 #if KERNEL_USE_EVENTFLAG
00029
00030 //---------------------------------------------------------------------------
00046 class EventFlag : public BlockingObject
00047 {
00048 public:
00052     void Init() { m_usSetMask = 0; m_clBlockList.Init(); }
00053
00061     K_USHORT Wait(K_USHORT usMask_, EventFlagOperation_t eMode_);
00062
00063 #if KERNEL_USE_TIMEOUTS
00064
00072     K_USHORT Wait(K_USHORT usMask_, EventFlagOperation_t eMode_, K_ULONG ulTimeMS_);
00073
00074     void WakeMe(Thread *pclOwner_);
00075
00076 #endif
00077
00083     void Set(K_USHORT usMask_);
00084
00089     void Clear(K_USHORT usMask_);
00090
00095     K_USHORT GetMask();
00096
00097 private:
00098
00099 #if KERNEL_USE_TIMEOUTS
00100     K_USHORT Wait_i(K_USHORT usMask_, EventFlagOperation_t eMode_, K_ULONG ulTimeMS_);
00101 #else
00102     K_USHORT Wait_i(K_USHORT usMask_, EventFlagOperation_t eMode_);
00103 #endif
00104
00105     K_USHORT m_usSetMask;
00106 };
00107
00108 #endif //KERNEL_USE_EVENTFLAG
00109 #endif //__EVENTFLAG_H__
00110
```

15.19 kernel.cpp File Reference

Kernel initialization and startup code.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kernel.h"
#include "scheduler.h"
#include "thread.h"
#include "threadport.h"
#include "timerlist.h"
#include "message.h"
#include "driver.h"
#include "profile.h"
#include "kprofile.h"
#include "tracebuffer.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** KERNEL_CPP

15.19.1 Detailed Description

Kernel initialization and startup code.

Definition in file kernel.cpp.

15.20 kernel.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____
00003  ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___   |
00005 |     \/   | ||     \      ||      \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023
00024 #include "kernel.h"
00025 #include "scheduler.h"
00026 #include "thread.h"
00027 #include "threadport.h"
00028 #include "timerlist.h"
00029 #include "message.h"
00030 #include "driver.h"
00031 #include "profile.h"
00032 #include "kprofile.h"
00033 #include "tracebuffer.h"
00034 #include "kernel_debug.h"
00035
00036 bool Kernel::m_bIsStarted;
00037 bool Kernel::m_bIsPanic;
00038 panic_func_t Kernel::m_pfPanic;
00039
00040 //---------------------------------------------------------------------------
00041 #if defined __FILE_ID__
00042     #undef __FILE_ID__
00043 #endif
00044 #define __FILE_ID__     KERNEL_CPP
00045
00046 //---------------------------------------------------------------------------
00047 void Kernel::Init(void)
00048 {
00049     m_bIsStarted = false;
00050     m_bIsPanic = false;
```

```
00051     m_pfPanic = 0;
00052
00053 #if KERNEL_USE_DEBUG
00054     TraceBuffer::Init();
00055 #endif
00056     KERNEL_TRACE( STR_MARK3_INIT );
00057
00058     // Initialize the global kernel data - scheduler, timer-scheduler, and
00059     // the global message pool.
00060     Scheduler::Init();
00061 #if KERNEL_USE_DRIVER
00062     DriverList::Init();
00063 #endif
00064 #if KERNEL_USE_TIMERS
00065     TimerScheduler::Init();
00066 #endif
00067 #if KERNEL_USE_MESSAGE
00068     GlobalMessagePool::Init();
00069 #endif
00070 #if KERNEL_USE_PROFILER
00071     Profiler::Init();
00072 #endif
00073 }
00074
00075 //---------------------------------------------------------------------------
00076 void Kernel::Start(void)
00077 {
00078     KERNEL_TRACE( STR_THREAD_START );
00079     m_bIsStarted = true;
00080     ThreadPort::StartThreads();
00081     KERNEL_TRACE( STR_START_ERROR );
00082 }
00083
00084 //---------------------------------------------------------------------------
00085 void Kernel::Panic(K_USHORT usCause_)
00086 {
00087     m_bIsPanic = true;
00088     if (m_pfPanic)
00089     {
00090         m_pfPanic(usCause_);
00091     }
00092     else
00093     {
00094 #if KERNEL_AWARE_SIMULATION
00095         Kernel_Aware::Exit_Simulator();
00096 #endif
00097         while(1);
00098     }
00099 }
```

15.21 kernel.h File Reference

Kernel initialization and startup class.

```
#include "kerneltypes.h"
#include "panic_codes.h"
```

Classes

- class Kernel

  Class that encapsulates all of the kernel startup functions.

15.21.1 Detailed Description

Kernel initialization and startup class. The Kernel namespace provides functions related to initializing and starting up the kernel.

The Kernel::Init() function must be called before any of the other functions in the kernel can be used.

Once the initial kernel configuration has been completed (i.e. first threads have been added to the scheduler), the Kernel::Start() function can then be called, which will transition code execution from the "main()" context to the

threads in the scheduler.

Definition in file kernel.h.

15.22 kernel.h

```
00001 /*=====================================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___   |
00005 |     \/   | ||     \      ||      \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]-----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====================================================================================*/
00032 #ifndef __KERNEL_H__
00033 #define __KERNEL_H__
00034
00035 #include "kerneltypes.h"
00036 #include "panic_codes.h"
00037
00038 //---------------------------------------------------------------------------
00042 class Kernel
00043 {
00044 public:
00053     static void Init(void);
00054
00067     static void Start(void);
00068
00074     static bool IsStarted()    {   return m_bIsStarted;    }
00075
00083     static void SetPanic( panic_func_t pfPanic_ ) { m_pfPanic = pfPanic_; }
00084
00089     static bool IsPanic()      {   return m_bIsPanic;   }
00090
00095     static void Panic(K_USHORT usCause_);
00096
00097 private:
00098     static bool m_bIsStarted;
00099     static bool m_bIsPanic;
00100     static panic_func_t m_pfPanic;
00101 };
00102
00103 #endif
00104
```

15.23 kernel_debug.h File Reference

Macros and functions used for assertions, kernel traces, etc.

```
#include "debug_tokens.h"
#include "mark3cfg.h"
#include "tracebuffer.h"
#include "kernel_aware.h"
#include "panic_codes.h"
#include "kernel.h"
```

Macros

- #define **__FILE_ID__** 0
- #define **KERNEL_TRACE**(x)
- #define **KERNEL_TRACE_1**(x, arg1)
- #define **KERNEL_TRACE_2**(x, arg1, arg2)
- #define **KERNEL_ASSERT**(x)

15.23.1 Detailed Description

Macros and functions used for assertions, kernel traces, etc.

Definition in file kernel_debug.h.

15.24 kernel_debug.h

```
00001 /*=========================================================================
00002      _____          _____          _____          _____
00003  ___|     |__   ___|_    |__   ___|__   |__   ___| __  |__   ______
00004 |    \   /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/    | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========================================================================*/
00020 #ifndef __KERNEL_DEBUG_H__
00021 #define __KERNEL_DEBUG_H__
00022
00023 #include "debug_tokens.h"
00024 #include "mark3cfg.h"
00025 #include "tracebuffer.h"
00026 #include "kernel_aware.h"
00027 #include "panic_codes.h"
00028 #include "kernel.h"
00029 //---------------------------------------------------------------------------
00030 #if (KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION)
00031
00032 //---------------------------------------------------------------------------
00033 #define __FILE_ID__         STR_UNDEFINED
00034
00035 //---------------------------------------------------------------------------
00036 #define KERNEL_TRACE( x )   \
00037 {   \
00038     K_USHORT ausMsg__[5]; \
00039     ausMsg__[0] = 0xACDC;  \
00040     ausMsg__[1] = __FILE_ID__; \
00041     ausMsg__[2] = __LINE__; \
00042     ausMsg__[3] = TraceBuffer::Increment() ; \
00043     ausMsg__[4] = (K_USHORT)(x) ; \
00044     TraceBuffer::Write(ausMsg__, 5); \
00045 };
00046
00047 //---------------------------------------------------------------------------
00048 #define KERNEL_TRACE_1( x, arg1 ) \
00049 {   \
00050     K_USHORT ausMsg__[6]; \
00051     ausMsg__[0] = 0xACDC;  \
00052     ausMsg__[1] = __FILE_ID__; \
00053     ausMsg__[2] = __LINE__; \
00054     ausMsg__[3] = TraceBuffer::Increment(); \
00055     ausMsg__[4] = (K_USHORT)(x); \
00056     ausMsg__[5] = arg1; \
00057     TraceBuffer::Write(ausMsg__, 6); \
00058 }
00059
00060 //---------------------------------------------------------------------------
00061 #define KERNEL_TRACE_2( x, arg1, arg2 ) \
00062 {   \
00063     K_USHORT ausMsg__[7]; \
00064     ausMsg__[0] = 0xACDC;  \
00065     ausMsg__[1] = __FILE_ID__; \
00066     ausMsg__[2] = __LINE__; \
00067     ausMsg__[3] = TraceBuffer::Increment(); \
00068     ausMsg__[4] = (K_USHORT)(x); \
00069     ausMsg__[5] = arg1; \
00070     ausMsg__[6] = arg2; \
00071     TraceBuffer::Write(ausMsg__, 7); \
00072 }
00073
00074 //---------------------------------------------------------------------------
00075 #define KERNEL_ASSERT( x ) \
00076 {       \
00077     if( ( x ) == false ) \
00078     {   \
00079         K_USHORT ausMsg__[5];   \
00080         ausMsg__[0] = 0xACDC;   \
```

```
00081         ausMsg__[1] = __FILE_ID__;  \
00082         ausMsg__[2] = __LINE__; \
00083         ausMsg__[3] = TraceBuffer::Increment(); \
00084         ausMsg__[4] = STR_ASSERT_FAILED;    \
00085         TraceBuffer::Write(ausMsg__, 5); \
00086         Kernel::Panic(PANIC_ASSERT_FAILED); \
00087     }   \
00088 }
00089
00090 #elif (KERNEL_USE_DEBUG && KERNEL_AWARE_SIMULATION)
00091 //---------------------------------------------------------------------------
00092 #define __FILE_ID__         STR_UNDEFINED
00093
00094 //---------------------------------------------------------------------------
00095 #define KERNEL_TRACE( x )   \
00096 {   \
00097     Kernel_Aware::Trace( __FILE_ID__, __LINE__, x ); \
00098 };
00099
00100 //---------------------------------------------------------------------------
00101 #define KERNEL_TRACE_1( x, arg1 ) \
00102 {   \
00103     Kernel_Aware::Trace( __FILE_ID__, __LINE__, x, arg1 ); \
00104 }
00105
00106 //---------------------------------------------------------------------------
00107 #define KERNEL_TRACE_2( x, arg1, arg2 ) \
00108 {   \
00109     Kernel_Aware::Trace( __FILE_ID__, __LINE__, x, arg1, arg2 ); \
00110 }
00111
00112 //---------------------------------------------------------------------------
00113 #define KERNEL_ASSERT( x ) \
00114 {       \
00115     if( ( x ) == false ) \
00116     {   \
00117         Kernel_Aware::Trace( __FILE_ID__, __LINE__, STR_ASSERT_FAILED ); \
00118         Kernel::Panic( PANIC_ASSERT_FAILED ); \
00119     }   \
00120 }
00121
00122 #else
00123 //---------------------------------------------------------------------------
00124 #define __FILE_ID__         0
00125 //---------------------------------------------------------------------------
00126 #define KERNEL_TRACE( x )
00127 //---------------------------------------------------------------------------
00128 #define KERNEL_TRACE_1( x, arg1 )
00129 //---------------------------------------------------------------------------
00130 #define KERNEL_TRACE_2( x, arg1, arg2 )
00131 //---------------------------------------------------------------------------
00132 #define KERNEL_ASSERT( x )
00133
00134 #endif // KERNEL_USE_DEBUG
00135
00136 #endif
```

15.25 kernelswi.cpp File Reference

Kernel Software interrupt implementation for ATMega328p.

```
#include "kerneltypes.h"
#include "kernelswi.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

15.25.1 Detailed Description

Kernel Software interrupt implementation for ATMega328p.

Definition in file kernelswi.cpp.

15.26 kernelswi.cpp

```
00001 /*=============================================================================
00002      _____        _____        _____        _____
00003  ___|     |__  __|_    |__  __|__   |__  __| __  |__  _______
00004 |     \   /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |      \/    | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =============================================================================*/
00022 #include "kerneltypes.h"
00023 #include "kernelswi.h"
00024
00025 #include <avr/io.h>
00026 #include <avr/interrupt.h>
00027
00028 //---------------------------------------------------------------------------
00029 void KernelSWI::Config(void)
00030 {
00031     PORTD &= ~0x04; // Clear INT0
00032     DDRD |= 0x04;    // Set PortD, bit 2 (INT0) As Output
00033     EICRA |= (1 << ISC00) | (1 << ISC01);    // Rising edge on INT0
00034 }
00035
00036 //---------------------------------------------------------------------------
00037 void KernelSWI::Start(void)
00038 {
00039     EIFR &= ~(1 << INTF0);    // Clear any pending interrupts on INT0
00040     EIMSK |= (1 << INT0);    // Enable INT0 interrupt (as K_LONG as I-bit is set)
00041 }
00042
00043 //---------------------------------------------------------------------------
00044 void KernelSWI::Stop(void)
00045 {
00046     EIMSK &= ~(1 << INT0);    // Disable INT0 interrupts
00047 }
00048
00049 //---------------------------------------------------------------------------
00050 K_UCHAR KernelSWI::DI()
00051 {
00052     bool bEnabled = ((EIMSK & (1 << INT0)) != 0);
00053     EIMSK &= ~(1 << INT0);
00054     return bEnabled;
00055 }
00056
00057 //---------------------------------------------------------------------------
00058 void KernelSWI::RI(bool bEnable_)
00059 {
00060     if (bEnable_)
00061     {
00062         EIMSK |= (1 << INT0);
00063     }
00064     else
00065     {
00066         EIMSK &= ~(1 << INT0);
00067     }
00068 }
00069
00070 //---------------------------------------------------------------------------
00071 void KernelSWI::Clear(void)
00072 {
00073     EIFR &= ~(1 << INTF0);     // Clear the interrupt flag for INT0
00074 }
00075
00076 //---------------------------------------------------------------------------
00077 void KernelSWI::Trigger(void)
00078 {
00079     //if(Thread_IsSchedulerEnabled())
00080     {
00081         PORTD &= ~0x04;
00082         PORTD |= 0x04;
00083     }
00084 }
```

15.27 kernelswi.h File Reference

Kernel Software interrupt declarations.

```
#include "kerneltypes.h"
```

Classes

- class KernelSWI

  Class providing the software-interrupt required for context-switching in the kernel.

15.27.1 Detailed Description

Kernel Software interrupt declarations.

Definition in file kernelswi.h.

15.28 kernelswi.h

```
00001 /*=========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========================================================================*/
00023 #include "kerneltypes.h"
00024 #ifndef __KERNELSWI_H_
00025 #define __KERNELSWI_H_
00026
00027 //---------------------------------------------------------------------------
00032 class KernelSWI
00033 {
00034 public:
00041     static void Config(void);
00042
00048     static void Start(void);
00049
00055     static void Stop(void);
00056
00062     static void Clear(void);
00063
00069     static void Trigger(void);
00070
00078     static K_UCHAR DI();
00079
00087     static void RI(bool bEnable_);
00088 };
00089
00090
00091 #endif // __KERNELSIW_H_
```

15.29 kerneltimer.cpp File Reference

Kernel Timer Implementation for ATMega328p.

```
#include "kerneltypes.h"
#include "kerneltimer.h"
#include "mark3cfg.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

Macros

- #define **TCCR1B_INIT** ((1 << WGM12) | (1 << CS12))
- #define **TIMER_IMSK** (1 << OCIE1A)
- #define **TIMER_IFR** (1 << OCF1A)

15.29.1 Detailed Description

Kernel Timer Implementation for ATMega328p.

Definition in file kerneltimer.cpp.

15.30 kerneltimer.cpp

```
00001 /*=============================================================================
00002       _____          _____          _____          _____
00003  ____|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \   /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  ___||__|\__\  ___||__|\__\  ___||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00021 #include "kerneltypes.h"
00022 #include "kerneltimer.h"
00023 #include "mark3cfg.h"
00024
00025 #include <avr/io.h>
00026 #include <avr/interrupt.h>
00027
00028 #define TCCR1B_INIT        ((1 << WGM12) | (1 << CS12))
00029 #define TIMER_IMSK         (1 << OCIE1A)
00030 #define TIMER_IFR          (1 << OCF1A)
00031
00032 //---------------------------------------------------------------------------
00033 void KernelTimer::Config(void)
00034 {
00035     TCCR1B = TCCR1B_INIT;
00036 }
00037
00038 //---------------------------------------------------------------------------
00039 void KernelTimer::Start(void)
00040 {
00041 #if !KERNEL_TIMERS_TICKLESS
00042     TCCR1B = ((1 << WGM12) | (1 << CS11) | (1 << CS10));
00043     OCR1A = ((SYSTEM_FREQ / 1000) / 64);
00044 #else
00045     TCCR1B |= (1 << CS12);
00046 #endif
00047
00048     TCNT1 = 0;
00049     TIFR1 &= ~TIMER_IFR;
00050     TIMSK1 |= TIMER_IMSK;
00051 }
00052
00053 //---------------------------------------------------------------------------
00054 void KernelTimer::Stop(void)
00055 {
00056 #if KERNEL_TIMERS_TICKLESS
00057     TIFR1 &= ~TIMER_IFR;
00058     TIMSK1 &= ~TIMER_IMSK;
00059     TCCR1B &= ~(1 << CS12);      // Disable count...
00060     TCNT1 = 0;
00061     OCR1A = 0;
00062 #endif
00063 }
00064
00065 //---------------------------------------------------------------------------
00066 K_USHORT KernelTimer::Read(void)
00067 {
00068 #if KERNEL_TIMERS_TICKLESS
00069     volatile K_USHORT usRead1;
00070     volatile K_USHORT usRead2;
```

```
00071
00072     do {
00073         usRead1 = TCNT1;
00074         usRead2 = TCNT1;
00075     } while (usRead1 != usRead2);
00076
00077     return usRead1;
00078 #else
00079     return 0;
00080 #endif
00081 }
00082
00083 //---------------------------------------------------------------------------
00084 K_ULONG KernelTimer::SubtractExpiry(K_ULONG ulInterval_)
00085 {
00086 #if KERNEL_TIMERS_TICKLESS
00087     OCR1A -= (K_USHORT)ulInterval_;
00088     return (K_ULONG)OCR1A;
00089 #else
00090     return 0;
00091 #endif
00092 }
00093
00094 //---------------------------------------------------------------------------
00095 K_ULONG KernelTimer::TimeToExpiry(void)
00096 {
00097 #if KERNEL_TIMERS_TICKLESS
00098     K_USHORT usRead = KernelTimer::Read();
00099     K_USHORT usOCR1A = OCR1A;
00100
00101     if (usRead >= usOCR1A)
00102     {
00103         return 0;
00104     }
00105     else
00106     {
00107         return (K_ULONG)(usOCR1A - usRead);
00108     }
00109 #else
00110     return 0;
00111 #endif
00112 }
00113
00114 //---------------------------------------------------------------------------
00115 K_ULONG KernelTimer::GetOvertime(void)
00116 {
00117     return KernelTimer::Read();
00118 }
00119
00120 //---------------------------------------------------------------------------
00121 K_ULONG KernelTimer::SetExpiry(K_ULONG ulInterval_)
00122 {
00123 #if KERNEL_TIMERS_TICKLESS
00124     K_USHORT usSetInterval;
00125     if (ulInterval_ > 65535)
00126     {
00127         usSetInterval = 65535;
00128     }
00129     else
00130     {
00131         usSetInterval = (K_USHORT)ulInterval_ ;
00132     }
00133     OCR1A = usSetInterval;
00134     return (K_ULONG)usSetInterval;
00135 #else
00136     return 0;
00137 #endif
00138 }
00139
00140 //---------------------------------------------------------------------------
00141 void KernelTimer::ClearExpiry(void)
00142 {
00143 #if KERNEL_TIMERS_TICKLESS
00144     OCR1A = 65535;                    // Clear the compare value
00145 #endif
00146 }
00147
00148 //---------------------------------------------------------------------------
00149 K_UCHAR KernelTimer::DI(void)
00150 {
00151 #if KERNEL_TIMERS_TICKLESS
00152     bool bEnabled = ((TIMSK1 & (TIMER_IMSK)) != 0);
00153     TIFR1 &= ~TIMER_IFR;      // Clear interrupt flags
00154     TIMSK1 &= ~TIMER_IMSK;    // Disable interrupt
00155     return bEnabled;
00156 #else
00157     return 0;
```

```
00158 #endif
00159 }
00160
00161 //---------------------------------------------------------------------------
00162 void KernelTimer::EI(void)
00163 {
00164     KernelTimer::RI(0);
00165 }
00166
00167 //---------------------------------------------------------------------------
00168 void KernelTimer::RI(bool bEnable_)
00169 {
00170 #if KERNEL_TIMERS_TICKLESS
00171     if (bEnable_)
00172     {
00173         TIMSK1 |= (1 << OCIE1A);    // Enable interrupt
00174     }
00175     else
00176     {
00177         TIMSK1 &= ~(1 << OCIE1A);
00178     }
00179 #endif
00180 }
```

15.31 kerneltimer.h File Reference

Kernel Timer Class declaration.

```
#include "kerneltypes.h"
```

Classes

- class KernelTimer

  Hardware timer interface, used by all scheduling/timer subsystems.

Macros

- #define **SYSTEM_FREQ** ((K_ULONG)16000000)
- #define **TIMER_FREQ** ((K_ULONG)(SYSTEM_FREQ / 256))

15.31.1 Detailed Description

Kernel Timer Class declaration.

Definition in file kerneltimer.h.

15.32 kerneltimer.h

```
00001 /*============================================================================
00002       _____          _____          _____          _____
00003  ____|    _|__   ___|_    |__   ___|__   |__   ___| __   |__   ______
00004 |    \   /   | ||     \       ||      |      ||   |/ /      ||___    |
00005 |     \/    | ||      \      ||      \      ||      \      ||___    |
00006 |__/\___/|__|_||__|\___\  ___||__|\___\  ___||__|\___\  ___||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ============================================================================*/
00021 #include "kerneltypes.h"
00022 #ifndef __KERNELTIMER_H_
00023 #define __KERNELTIMER_H_
00024
```

```
00025 //---------------------------------------------------------------------------
00026 #define SYSTEM_FREQ         ((K_ULONG)16000000)
00027 #define TIMER_FREQ        ((K_ULONG)(SYSTEM_FREQ / 256)) // Timer ticks per second...
00028
00029 //---------------------------------------------------------------------------
00033 class KernelTimer
00034 {
00035 public:
00041     static void Config(void);
00042
00048     static void Start(void);
00049
00055     static void Stop(void);
00056
00062     static K_UCHAR DI(void);
00063
00071     static void RI(bool bEnable_);
00072
00078     static void EI(void);
00079
00090     static K_ULONG SubtractExpiry(K_ULONG ulInterval_);
00091
00100     static K_ULONG TimeToExpiry(void);
00101
00110     static K_ULONG SetExpiry(K_ULONG ulInterval_);
00111
00120     static K_ULONG GetOvertime(void);
00121
00127     static void ClearExpiry(void);
00128
00129 private:
00137     static K_USHORT Read(void);
00138
00139 };
00140
00141 #endif //__KERNELTIMER_H_
```

15.33 kerneltypes.h File Reference

Basic data type primatives used throughout the OS.

`#include <stdint.h>`

Macros

- #define **K_BOOL** uint8_t
- #define **K_CHAR** char
- #define **K_UCHAR** uint8_t
- #define **K_USHORT** uint16_t
- #define **K_SHORT** int16_t
- #define **K_ULONG** uint32_t
- #define **K_LONG** int32_t
- #define **K_ADDR** uint16_t
- #define **K_WORD** uint8_t

Typedefs

- typedef void(∗ **panic_func_t**)(K_USHORT usPanicCode_)

Enumerations

- enum **EventFlagOperation_t** {
  EVENT_FLAG_ALL, **EVENT_FLAG_ANY**, **EVENT_FLAG_ALL_CLEAR**, **EVENT_FLAG_ANY_CLEAR**,
  EVENT_FLAG_MODES, **EVENT_FLAG_PENDING_UNBLOCK** }

15.33.1 Detailed Description

Basic data type primatives used throughout the OS.

Definition in file kerneltypes.h.

15.34 kerneltypes.h

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   __|_    |__   __|__   |__   __| __  |__  _______
00004 |    \  /  | ||    \       ||      |     ||   |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \      ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00019 #include <stdint.h>
00020
00021 #ifndef __KERNELTYPES_H__
00022 #define __KERNELTYPES_H__
00023
00024 #if defined(bool)
00025     #define K_BOOL            bool
00026 #else
00027     #define K_BOOL            uint8_t
00028 #endif
00029
00030 #define K_CHAR          char
00031 #define K_UCHAR         uint8_t
00032 #define K_USHORT        uint16_t
00033 #define K_SHORT         int16_t
00034 #define K_ULONG         uint32_t
00035 #define K_LONG          int32_t
00036
00037 #define K_ADDR      uint16_t
00038 #define K_WORD      uint8_t
00039
00040 //---------------------------------------------------------------------------
00041 typedef void (*panic_func_t)( K_USHORT usPanicCode_ );
00042
00043 //---------------------------------------------------------------------------
00044 typedef enum
00045 {
00046     EVENT_FLAG_ALL,
00047     EVENT_FLAG_ANY,
00048     EVENT_FLAG_ALL_CLEAR,
00049     EVENT_FLAG_ANY_CLEAR,
00050     EVENT_FLAG_MODES,
00051     EVENT_FLAG_PENDING_UNBLOCK
00052 } EventFlagOperation_t;
00053
00054
00055 #endif
```

15.35 kprofile.cpp File Reference

ATMega328p Profiling timer implementation.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "profile.h"
#include "kprofile.h"
#include "threadport.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

15.35.1 Detailed Description

ATMega328p Profiling timer implementation.

Definition in file kprofile.cpp.

15.36 kprofile.cpp

```
00001 /*=========================================================================
00002      _____          _____          _____          _____
00003  ___|     |_|__  __|_    |__  __|__    |__  __| __  |__  _______
00004 |    \   /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/    | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========================================================================*/
00020 #include "kerneltypes.h"
00021 #include "mark3cfg.h"
00022 #include "profile.h"
00023 #include "kprofile.h"
00024 #include "threadport.h"
00025 #include <avr/io.h>
00026 #include <avr/interrupt.h>
00027
00028 #if KERNEL_USE_PROFILER
00029 K_ULONG Profiler::m_ulEpoch;
00030
00031 //---------------------------------------------------------------------------
00032 void Profiler::Init()
00033 {
00034     TCCR0A = 0;
00035     TCCR0B = 0;
00036     TIFR0 = 0;
00037     TIMSK0 = 0;
00038     m_ulEpoch = 0;
00039 }
00040
00041 //---------------------------------------------------------------------------
00042 void Profiler::Start()
00043 {
00044     TIFR0 = 0;
00045     TCNT0 = 0;
00046     TCCR0B |= (1 << CS01);
00047     TIMSK0 |= (1 << TOIE0);
00048 }
00049
00050 //---------------------------------------------------------------------------
00051 void Profiler::Stop()
00052 {
00053     TIFR0 = 0;
00054     TCCR0B &= ~(1 << CS01);
00055     TIMSK0 &= ~(1 << TOIE0);
00056 }
00057 //---------------------------------------------------------------------------
00058 K_USHORT Profiler::Read()
00059 {
00060     K_USHORT usRet;
00061     CS_ENTER();
00062     TCCR0B &= ~(1 << CS01);
00063     usRet = TCNT0;
00064     TCCR0B |= (1 << CS01);
00065     CS_EXIT();
00066     return usRet;
00067 }
00068
00069 //---------------------------------------------------------------------------
00070 void Profiler::Process()
00071 {
00072     CS_ENTER();
00073     m_ulEpoch++;
00074     CS_EXIT();
00075 }
00076
00077 //---------------------------------------------------------------------------
00078 ISR(TIMER0_OVF_vect)
00079 {
00080     Profiler::Process();
```

```
00081 }
00082
00083 #endif
```

15.37 kprofile.h File Reference

Profiling timer hardware interface.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
```

15.37.1 Detailed Description

Profiling timer hardware interface.

Definition in file kprofile.h.

15.38 kprofile.h

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   __|_    |__   __|__   |__   __| __  |__  _______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00020 #include "kerneltypes.h"
00021 #include "mark3cfg.h"
00022 #include "ll.h"
00023
00024 #ifndef __KPROFILE_H__
00025 #define __KPROFILE_H__
00026
00027 #if KERNEL_USE_PROFILER
00028
00029 //---------------------------------------------------------------------------
00030 #define TICKS_PER_OVERFLOW              (256)
00031 #define CLOCK_DIVIDE                    (8)
00032
00033 //---------------------------------------------------------------------------
00037 class Profiler
00038 {
00039 public:
00046     static void Init();
00047
00053     static void Start();
00054
00060     static void Stop();
00061
00067     static K_USHORT Read();
00068
00072     static void Process();
00073
00077     static K_ULONG GetEpoch(){ return m_ulEpoch; }
00078 private:
00079
00080     static K_ULONG m_ulEpoch;
00081 };
00082
00083 #endif //KERNEL_USE_PROFILER
00084
00085 #endif
00086
```

15.39 ksemaphore.cpp File Reference

Semaphore Blocking-Object Implemenation.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ksemaphore.h"
#include "blocking.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** SEMAPHORE_CPP

15.39.1 Detailed Description

Semaphore Blocking-Object Implemenation.

Definition in file ksemaphore.cpp.

15.40 ksemaphore.cpp

```
00001 /*==========================================================================
00002      ______          ______          ______          ______
00003  ___|     _|__   __|_     |__   __|__     |__   __| __   |__   _______
00004 |     \   /  | ||     \       ||      |      ||   |/ /      ||___    |
00005 |      \/    | ||      \      ||      \      ||      \      ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |______|       |______|       |______|       |______|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ==========================================================================*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "ksemaphore.h"
00026 #include "blocking.h"
00027 #include "kernel_debug.h"
00028 //---------------------------------------------------------------------------
00029 #if defined __FILE_ID__
00030     #undef __FILE_ID__
00031 #endif
00032 #define __FILE_ID__     SEMAPHORE_CPP
00033
00034 #if KERNEL_USE_SEMAPHORE
00035
00036 #if KERNEL_USE_TIMEOUTS
00037 #include "timerlist.h"
00038
00039 //---------------------------------------------------------------------------
00040 void TimedSemaphore_Callback(Thread *pclOwner_, void *pvData_)
00041 {
00042     Semaphore *pclSemaphore = static_cast<Semaphore*>(pvData_);
00043
00044     // Indicate that the semaphore has expired on the thread
00045     pclOwner_->SetExpired(true);
00046
00047     // Wake up the thread that was blocked on this semaphore.
00048     pclSemaphore->WakeMe(pclOwner_);
00049
00050     if (pclOwner_->GetPriority() > Scheduler::GetCurrentThread()->
      GetPriority())
00051     {
00052         Thread::Yield();
00053     }
00054 }
00055
00056 //---------------------------------------------------------------------------
```

```
00057 void Semaphore::WakeMe(Thread *pclChosenOne_)
00058 {
00059     // Remove from the semaphore waitlist and back to its ready list.
00060     UnBlock(pclChosenOne_);
00061 }
00062
00063 #endif // KERNEL_USE_TIMEOUTS
00064
00065 //---------------------------------------------------------------------------
00066 K_UCHAR Semaphore::WakeNext()
00067 {
00068     Thread *pclChosenOne;
00069
00070     pclChosenOne = m_clBlockList.HighestWaiter();
00071
00072     // Remove from the semaphore waitlist and back to its ready list.
00073     UnBlock(pclChosenOne);
00074
00075     // Call a task switch only if higher priority thread
00076     if (pclChosenOne->GetPriority() > Scheduler::GetCurrentThread()->
      GetPriority())
00077     {
00078         return 1;
00079     }
00080     return 0;
00081 }
00082
00083 //---------------------------------------------------------------------------
00084 void Semaphore::Init(K_USHORT usInitVal_, K_USHORT usMaxVal_)
00085 {
00086     // Copy the paramters into the object - set the maximum value for this
00087     // semaphore to implement either binary or counting semaphores, and set
00088     // the initial count.  Clear the wait list for this object.
00089     m_usValue = usInitVal_;
00090     m_usMaxValue = usMaxVal_;
00091
00092     m_clBlockList.Init();
00093 }
00094
00095 //---------------------------------------------------------------------------
00096 bool Semaphore::Post()
00097 {
00098     KERNEL_TRACE_1( STR_SEMAPHORE_POST_1, (K_USHORT)g_pstCurrent->GetID() );
00099
00100     bool bThreadWake = 0;
00101     K_BOOL bBail = false;
00102     // Increment the semaphore count - we can mess with threads so ensure this
00103     // is in a critical section.  We don't just disable the scheudler since
00104     // we want to be able to do this from within an interrupt context as well.
00105     CS_ENTER();
00106
00107     // If nothing is waiting for the semaphore
00108     if (m_clBlockList.GetHead() == NULL)
00109     {
00110         // Check so see if we've reached the maximum value in the semaphore
00111         if (m_usValue < m_usMaxValue)
00112         {
00113             // Increment the count value
00114             m_usValue++;
00115         }
00116         else
00117         {
00118             // Maximum value has been reached, bail out.
00119             bBail = true;
00120         }
00121     }
00122     else
00123     {
00124         // Otherwise, there are threads waiting for the semaphore to be
00125         // posted, so wake the next one (highest priority goes first).
00126         bThreadWake = WakeNext();
00127     }
00128
00129     CS_EXIT();
00130
00131     // If we weren't able to increment the semaphore count, fail out.
00132     if (bBail)
00133     {
00134         return false;
00135     }
00136
00137     // if bThreadWake was set, it means that a higher-priority thread was
00138     // woken.  Trigger a context switch to ensure that this thread gets
00139     // to execute next.
00140     if (bThreadWake)
00141     {
00142         Thread::Yield();
```

```
00143     }
00144     return true;
00145 }
00146 
00147 //---------------------------------------------------------------------------
00148 #if KERNEL_USE_TIMEOUTS
00149 bool Semaphore::Pend_i( K_ULONG ulWaitTimeMS_ )
00150 #else
00151 void Semaphore::Pend_i( void )
00152 #endif
00153 {
00154     KERNEL_TRACE_1( STR_SEMAPHORE_PEND_1, (K_USHORT)g_pstCurrent->GetID() );
00155 
00156 #if KERNEL_USE_TIMEOUTS
00157     Timer clSemTimer;
00158     bool bUseTimer = false;
00159 #endif
00160 
00161     // Once again, messing with thread data - ensure
00162     // we're doing all of these operations from within a thread-safe context.
00163     CS_ENTER();
00164 
00165     // Check to see if we need to take any action based on the semaphore count
00166     if (m_usValue != 0)
00167     {
00168         // The semaphore count is non-zero, we can just decrement the count
00169         // and go along our merry way.
00170         m_usValue--;
00171     }
00172     else
00173     {
00174         // The semaphore count is zero - we need to block the current thread
00175         // and wait until the semaphore is posted from elsewhere.
00176 #if KERNEL_USE_TIMEOUTS
00177         if (ulWaitTimeMS_)
00178         {
00179             g_pstCurrent->SetExpired(false);
00180             clSemTimer.Init();
00181             clSemTimer.Start(0, ulWaitTimeMS_, TimedSemaphore_Callback, (void*)this);
00182             bUseTimer = true;
00183         }
00184 #endif
00185         Block(g_pstCurrent);
00186 
00187         // Switch Threads immediately
00188         Thread::Yield();
00189     }
00190 
00191     CS_EXIT();
00192 
00193 #if KERNEL_USE_TIMEOUTS
00194     if (bUseTimer)
00195     {
00196         clSemTimer.Stop();
00197         return (g_pstCurrent->GetExpired() == 0);
00198     }
00199     return true;
00200 #endif
00201 }
00202 
00203 //---------------------------------------------------------------------------
00204 // Redirect the untimed pend API to the timed pend, with a null timeout.
00205 void Semaphore::Pend()
00206 {
00207 #if KERNEL_USE_TIMEOUTS
00208     Pend_i(0);
00209 #else
00210     Pend_i();
00211 #endif
00212 }
00213 
00214 #if KERNEL_USE_TIMEOUTS
00215 //---------------------------------------------------------------------------
00216 bool Semaphore::Pend( K_ULONG ulWaitTimeMS_ )
00217 {
00218     return Pend_i( ulWaitTimeMS_ );
00219 }
00220 #endif
00221 
00222 //---------------------------------------------------------------------------
00223 K_USHORT Semaphore::GetCount()
00224 {
00225     K_USHORT usRet;
00226     CS_ENTER();
00227     usRet = m_usValue;
00228     CS_EXIT();
00229     return usRet;
```

```
00230 }
00231
00232 #endif
```

15.41 ksemaphore.h File Reference

Semaphore Blocking Object class declarations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
#include "threadlist.h"
```

Classes

- class Semaphore

  Counting semaphore, based on BlockingObject base class.

15.41.1 Detailed Description

Semaphore Blocking Object class declarations.

Definition in file ksemaphore.h.

15.42 ksemaphore.h

```
00001 /*============================================================================
00002     _____          _____          _____          _____
00003  ___|    _|__   __|_    |__   __|__   |__   __| __   |__  _______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007    |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ============================================================================*/
00022 #ifndef __KSEMAPHORE_H__
00023 #define __KSEMAPHORE_H__
00024
00025 #include "kerneltypes.h"
00026 #include "mark3cfg.h"
00027
00028 #include "blocking.h"
00029 #include "threadlist.h"
00030
00031 #if KERNEL_USE_SEMAPHORE
00032
00033 //---------------------------------------------------------------------------
00037 class Semaphore : public BlockingObject
00038 {
00039 public:
00049     void Init(K_USHORT usInitVal_, K_USHORT usMaxVal_);
00050
00059     bool Post();
00060
00067     void Pend();
00068
00069
00081     K_USHORT GetCount();
00082
00083 #if KERNEL_USE_TIMEOUTS
00084
00095     bool Pend( K_ULONG ulWaitTimeMS_);
00096
00107     void WakeMe(Thread *pclChosenOne_);
```

```
00108 #endif
00109
00110 private:
00111
00117     K_UCHAR WakeNext();
00118
00119
00120 #if KERNEL_USE_TIMEOUTS
00121     bool Pend_i( K_ULONG ulWaitTimeMS_ );
00122 #else
00123     void Pend_i( void );
00124 #endif
00125
00126     K_USHORT m_usValue;
00127     K_USHORT m_usMaxValue;
00128
00129
00130 };
00131
00132 #endif //KERNEL_USE_SEMAPHORE
00133
00134 #endif
```

15.43 ll.cpp File Reference

Core Linked-List implementation, from which all kernel objects are derived.

```
#include "kerneltypes.h"
#include "kernel.h"
#include "ll.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** LL_CPP

15.43.1 Detailed Description

Core Linked-List implementation, from which all kernel objects are derived.

Definition in file ll.cpp.

15.44 ll.cpp

```
00001 /*===================================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \  /   | ||     \       ||      |     ||   |/ /     ||___    |
00005 |     \/    | ||      \      ||      \     ||      \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===================================================================================*/
00022 #include "kerneltypes.h"
00023 #include "kernel.h"
00024 #include "ll.h"
00025 #include "kernel_debug.h"
00026
00027 //---------------------------------------------------------------------------
00028 #if defined __FILE_ID__
00029     #undef __FILE_ID__
00030 #endif
00031 #define __FILE_ID__     LL_CPP
00032
```

```
00033 //---------------------------------------------------------------------------
00034 void LinkListNode::ClearNode()
00035 {
00036     next = NULL;
00037     prev = NULL;
00038 }
00039
00040 //---------------------------------------------------------------------------
00041 void DoubleLinkList::Add(LinkListNode *node_)
00042 {
00043     KERNEL_ASSERT( node_ );
00044
00045     // Add a node to the end of the linked list.
00046     if (!m_pstHead)
00047     {
00048         // If the list is empty, initilize the nodes
00049         m_pstHead = node_;
00050         m_pstTail = node_;
00051
00052         m_pstHead->prev = NULL;
00053         m_pstTail->next = NULL;
00054         return;
00055     }
00056
00057     // Move the tail node, and assign it to the new node just passed in
00058     m_pstTail->next = node_;
00059     node_->prev = m_pstTail;
00060     node_->next = NULL;
00061     m_pstTail = node_;
00062 }
00063
00064 //---------------------------------------------------------------------------
00065 void DoubleLinkList::Remove(LinkListNode *node_)
00066 {
00067     KERNEL_ASSERT( node_ );
00068
00069     if (node_->prev)
00070     {
00071 #if SAFE_UNLINK
00072         if (node_->prev->next != node_)
00073         {
00074             Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00075         }
00076 #endif
00077         node_->prev->next = node_->next;
00078     }
00079     if (node_->next)
00080     {
00081 #if SAFE_UNLINK
00082         if (node_->next->prev != node_)
00083         {
00084             Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00085         }
00086 #endif
00087         node_->next->prev = node_->prev;
00088     }
00089     if (node_ == m_pstHead)
00090     {
00091         m_pstHead = node_->next;
00092     }
00093     if (node_ == m_pstTail)
00094     {
00095         m_pstTail = node_->prev;
00096     }
00097
00098     node_->ClearNode();
00099 }
00100
00101 //---------------------------------------------------------------------------
00102 void CircularLinkList::Add(LinkListNode *node_)
00103 {
00104     KERNEL_ASSERT( node_ );
00105
00106     // Add a node to the end of the linked list.
00107     if (!m_pstHead)
00108     {
00109         // If the list is empty, initilize the nodes
00110         m_pstHead = node_;
00111         m_pstTail = node_;
00112
00113         m_pstHead->prev = m_pstHead;
00114         m_pstHead->next = m_pstHead;
00115         return;
00116     }
00117
00118     // Move the tail node, and assign it to the new node just passed in
00119     m_pstTail->next = node_;
```

```
00120     node_->prev = m_pstTail;
00121     node_->next = m_pstHead;
00122     m_pstTail = node_;
00123     m_pstHead->prev = node_;
00124 }
00125
00126 //---------------------------------------------------------------------------
00127 void CircularLinkList::Remove(LinkListNode *node_)
00128 {
00129     KERNEL_ASSERT( node_ );
00130
00131     // Check to see if this is the head of the list...
00132     if ((node_ == m_pstHead) && (m_pstHead == m_pstTail))
00133     {
00134         // Clear the head and tail pointers - nothing else left.
00135         m_pstHead = NULL;
00136         m_pstTail = NULL;
00137         return;
00138     }
00139
00140 #if SAFE_UNLINK
00141     // Verify that all nodes are properly connected
00142     if ((node_->prev->next != node_) || (node_->next->prev != node_))
00143     {
00144         Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00145     }
00146 #endif
00147
00148     // This is a circularly linked list - no need to check for connection,
00149     // just remove the node.
00150     node_->next->prev = node_->prev;
00151     node_->prev->next = node_->next;
00152
00153     if (node_ == m_pstHead)
00154     {
00155         m_pstHead = m_pstHead->next;
00156     }
00157     if (node_ == m_pstTail)
00158     {
00159         m_pstTail = m_pstTail->prev;
00160     }
00161     node_->ClearNode();
00162 }
00163
00164 //---------------------------------------------------------------------------
00165 void CircularLinkList::PivotForward()
00166 {
00167     if (m_pstHead)
00168     {
00169         m_pstHead = m_pstHead->next;
00170         m_pstTail = m_pstTail->next;
00171     }
00172 }
00173
00174 //---------------------------------------------------------------------------
00175 void CircularLinkList::PivotBackward()
00176 {
00177     if (m_pstHead)
00178     {
00179         m_pstHead = m_pstHead->prev;
00180         m_pstTail = m_pstTail->prev;
00181     }
00182 }
```

15.45 ll.h File Reference

Core linked-list declarations, used by all kernel list types.

`#include "kerneltypes.h"`

Classes

- class LinkListNode

  Basic linked-list node data structure.
- class LinkList

  Abstract-data-type from which all other linked-lists are derived.

- class DoubleLinkList

  Doubly-linked-list data type, inherited from the base LinkList type.
- class CircularLinkList

  Circular-linked-list data type, inherited from the base LinkList type.

Macros

- #define **NULL** (0)

15.45.1 Detailed Description

Core linked-list declarations, used by all kernel list types. At the heart of RTOS data structures are linked lists. Having a robust and efficient set of linked-list types that we can use as a foundation for building the rest of our kernel types allows us to keep our RTOS code efficient and logically-separated.

So what data types rely on these linked-list classes?

-Threads -ThreadLists -The Scheduler -Timers, -The Timer Scheduler -Blocking objects (Semaphores, Mutexes, etc...)

Pretty much everything in the kernel uses these linked lists. By having objects inherit from the base linked-list node type, we're able to leverage the double and circular linked-list classes to manager virtually every object type in the system without duplicating code. These functions are very efficient as well, allowing for very deterministic behavior in our code.

Definition in file ll.h.

15.46 ll.h

```
00001 /*=====================================================================
00002      _____          _____          _____          _____
00003  ___|     |__   ___|_    |__   ___|__   |__   ___| __  |__  ______
00004 |    \   /  | ||     \       ||      |     ||   |/ /     ||___    |
00005 |     \/    | ||      \      ||      \     ||      \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====================================================================*/
00043 #ifndef __LL_H__
00044 #define __LL_H__
00045
00046 #include "kerneltypes.h"
00047
00048 //---------------------------------------------------------------------------
00049 #ifndef NULL
00050 #define NULL        (0)
00051 #endif
00052
00053 //---------------------------------------------------------------------------
00059 class LinkList;
00060 class DoubleLinkList;
00061 class CircularLinkList;
00062
00063 //---------------------------------------------------------------------------
00068 class LinkListNode
00069 {
00070 protected:
00071
00072     LinkListNode *next;
00073     LinkListNode *prev;
00074
00075     LinkListNode() { }
00076
00082     void ClearNode();
00083
00084 public:
```

```
00092     LinkListNode *GetNext(void) { return next; }
00093
00101     LinkListNode *GetPrev(void) { return prev; }
00102
00103     friend class LinkList;
00104     friend class DoubleLinkList;
00105     friend class CircularLinkList;
00106 };
00107
00108 //---------------------------------------------------------------------------
00112 class LinkList
00113 {
00114 protected:
00115     LinkListNode *m_pstHead;
00116     LinkListNode *m_pstTail;
00117
00118 public:
00122     void Init(){ m_pstHead = NULL; m_pstTail = NULL; }
00123
00131     virtual void Add(LinkListNode *node_) = 0;
00132
00140     virtual void Remove(LinkListNode *node_) = 0;
00141
00149     LinkListNode *GetHead() { return m_pstHead; }
00150
00158     LinkListNode *GetTail() { return m_pstTail; }
00159 };
00160
00161 //---------------------------------------------------------------------------
00165 class DoubleLinkList : public LinkList
00166 {
00167 public:
00171     DoubleLinkList() { m_pstHead = NULL; m_pstTail = NULL; }
00172
00180     virtual void Add(LinkListNode *node_);
00181
00189     virtual void Remove(LinkListNode *node_);
00190 };
00191
00192 //---------------------------------------------------------------------------
00196 class CircularLinkList : public LinkList
00197 {
00198 public:
00199     CircularLinkList() { m_pstHead = NULL; m_pstTail = NULL; }
00200
00208     virtual void Add(LinkListNode *node_);
00209
00217     virtual void Remove(LinkListNode *node_);
00218
00225     void PivotForward();
00226
00233     void PivotBackward();
00234 };
00235
00236 #endif
```

15.47 manual.h File Reference

Ascii-format documentation, used by doxygen to create various printable and viewable forms.

15.47.1 Detailed Description

Ascii-format documentation, used by doxygen to create various printable and viewable forms.

Definition in file manual.h.

15.48 manual.h

```
00001 /*=============================================================================
00002       _____          _____          _____          _____
00003  ____|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \   /   | ||     \       ||       |     ||   |/ /      ||____    |
00005 |     \/    | ||      \      ||       \     ||       \      ||____    |
00006 |__/\__/|__|_||__|\__\   __||__|\__\   __||__|\__\   __||_______|
00007     |_____|       |_____|       |_____|       |_____|
```

```
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
```

15.49 mark3cfg.h File Reference

Mark3 Kernel Configuration.

Macros

- #define KERNEL_USE_TIMERS (1)

  The following options is related to all kernel time-tracking.
- #define KERNEL_USE_QUANTUM (1)

  Do you want to enable time quanta? This is useful when you want to have tasks in the same priority group share time in a controlled way.
- #define KERNEL_USE_SEMAPHORE (1)

  Do you want the ability to use counting/binary semaphores for thread synchronization? Enabling this features provides fully-blocking semaphores and enables all API functions declared in semaphore.h.
- #define KERNEL_USE_MESSAGE (1)

  Enable inter-thread messaging using named mailboxes.
- #define GLOBAL_MESSAGE_POOL_SIZE (8)

  If Messages are enabled, define the size of the default kernel message pool.
- #define KERNEL_USE_MUTEX (1)

  Do you want the ability to use mutual exclusion semaphores (mutex) for resource/block protection? Enabling this feature provides mutexes, with priority inheritence, as declared in mutex.h.
- #define KERNEL_USE_SLEEP (1)

  Do you want to be able to set threads to sleep for a specified time? This enables the Thread::Sleep() API.
- #define KERNEL_USE_DRIVER (0)

  Enabling device drivers provides a posix-like filesystem interface for peripheral device drivers.
- #define KERNEL_USE_THREADNAME (1)

  Provide Thread method to allow the user to set a name for each thread in the system.
- #define KERNEL_USE_DYNAMIC_THREADS (1)

  Provide extra Thread methods to allow the application to create (and more importantly destroy) threads at runtime.
- #define KERNEL_USE_PROFILER (0)

  Provides extra classes for profiling the performance of code.
- #define KERNEL_USE_DEBUG (0)

  Provides extra logic for kernel debugging, and instruments the kernel with extra asserts, and kernel trace functionality.
- #define **KERNEL_USE_EVENTFLAG** (1)

15.49.1 Detailed Description

Mark3 Kernel Configuration. This file is used to configure the kernel for your specific application in order to provide the optimal set of features for a given use case.

Since you only pay the price (code space/RAM) for the features you use, you can usually find a sweet spot between features and resource usage by picking and choosing features a-la-carte. This config file is written in an "interactive" way, in order to minimize confusion about what each option provides, and to make dependencies obvious.

As of 7.6.2012 on AVR, these are the costs associated with the various features:

Base Kernel: 2888 bytes Tickless Timers: 1194 bytes Semaphores: 224 bytes Message Queues: 332 bytes (+ Semaphores) Mutexes: 290 bytes Thread Sleep: 162 bytes (+ Semaphores/Timers) Round-Robin: 304 bytes (+ Timers) Drivers: 144 bytes Dynamic Threads: 68 bytes Thread Names: 8 bytes Profiling Timers: 624 bytes

Definition in file mark3cfg.h.

15.49.2 Macro Definition Documentation

15.49.2.1 #define GLOBAL_MESSAGE_POOL_SIZE (8)

If Messages are enabled, define the size of the default kernel message pool.

Messages can be manually added to the message pool, but this mechansims is more convenient and automatic.

Definition at line 99 of file mark3cfg.h.

15.49.2.2 #define KERNEL_USE_DRIVER (0)

Enabling device drivers provides a posix-like filesystem interface for peripheral device drivers.

When enabled, the size of the filesystem table is specified in DRIVER_TABLE_SIZE. Permissions are enforced for driver access by thread ID and group when DRIVER_USE_PERMS are enabled.

Definition at line 127 of file mark3cfg.h.

15.49.2.3 #define KERNEL_USE_DYNAMIC_THREADS (1)

Provide extra Thread methods to allow the application to create (and more importantly destroy) threads at runtime.

Useful for designs implementing worker threads, or threads that can be restarted after encountering error conditions.

Definition at line 142 of file mark3cfg.h.

15.49.2.4 #define KERNEL_USE_MESSAGE (1)

Enable inter-thread messaging using named mailboxes.

If per-thread mailboxes are defined, each thread is allocated a default mailbox of a depth specified by THREAD_MAILBOX_SIZE.

Definition at line 88 of file mark3cfg.h.

15.49.2.5 #define KERNEL_USE_MUTEX (1)

Do you want the ability to use mutual exclusion semaphores (mutex) for resource/block protection? Enabling this feature provides mutexes, with priority inheritence, as declared in mutex.h.

Enabling per-thread mutex automatically allocates a mutex for each thread.

Definition at line 108 of file mark3cfg.h.

15.49.2.6 #define KERNEL_USE_PROFILER (0)

Provides extra classes for profiling the performance of code.

Useful for debugging and development, but uses an additional timer.

Definition at line 148 of file mark3cfg.h.

15.49.2.7 #define KERNEL_USE_QUANTUM (1)

Do you want to enable time quanta? This is useful when you want to have tasks in the same priority group share time in a controlled way.

This allows equal tasks to use unequal amounts of the CPU, which is a great way to set up CPU budgets per thread in a round-robin scheduling system. If enabled, you can specify a number of ticks that serves as the default time period (quantum). Unless otherwise specified, every thread in a priority will get the default quantum.

Definition at line 68 of file mark3cfg.h.

15.49.2.8 #define KERNEL_USE_SEMAPHORE (1)

Do you want the ability to use counting/binary semaphores for thread synchronization? Enabling this features provides fully-blocking semaphores and enables all API functions declared in semaphore.h.

If you have to pick one blocking mechanism, this is the one to choose. By also enabling per-thread semaphores, each thread will receive it's own built-in semaphore.

Definition at line 80 of file mark3cfg.h.

15.49.2.9 #define KERNEL_USE_THREADNAME (1)

Provide Thread method to allow the user to set a name for each thread in the system.

Adds to the size of the thread member data.

Definition at line 134 of file mark3cfg.h.

15.49.2.10 #define KERNEL_USE_TIMERS (1)

The following options is related to all kernel time-tracking.

-timers provide a way for events to be periodically triggered in a lightweight manner. These can be periodic, or one-shot.

-Thread Quantum (used for round-robin scheduling) is dependent on this module, as is Thread Sleep functionality.

Definition at line 56 of file mark3cfg.h.

15.50 mark3cfg.h

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__   |__   ___| __  |__   ______
00004 |    \  /   | ||     \       ||      |      ||   |/ /      ||___    |
00005 |     \/    | ||      \      ||      \      ||      \      ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]-----------------------------------------------
00010
00011 Copyright (c) 2012 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00044 #ifndef __MARK3CFG_H__
00045 #define __MARK3CFG_H__
00046
00056 #define KERNEL_USE_TIMERS               (1)
00057
00067 #if KERNEL_USE_TIMERS
00068     #define KERNEL_USE_QUANTUM          (1)
00069 #else
00070     #define KERNEL_USE_QUANTUM          (0)
00071 #endif
00072
00080 #define KERNEL_USE_SEMAPHORE            (1)
00081
```

```
00087 #if KERNEL_USE_SEMAPHORE
00088     #define KERNEL_USE_MESSAGE          (1)
00089 #else
00090     #define KERNEL_USE_MESSAGE          (0)
00091 #endif
00092
00098 #if KERNEL_USE_MESSAGE
00099     #define GLOBAL_MESSAGE_POOL_SIZE    (8)
00100 #endif
00101
00108 #define KERNEL_USE_MUTEX                (1)
00109
00114 #if KERNEL_USE_TIMERS && KERNEL_USE_SEMAPHORE
00115     #define KERNEL_USE_SLEEP            (1)
00116 #else
00117     #define KERNEL_USE_SLEEP            (0)
00118 #endif
00119
00120
00127 #define KERNEL_USE_DRIVER               (0)
00128
00134 #define KERNEL_USE_THREADNAME           (1)
00135
00142 #define KERNEL_USE_DYNAMIC_THREADS      (1)
00143
00148 #define KERNEL_USE_PROFILER             (0)
00149
00154 #define KERNEL_USE_DEBUG                (0)
00155
00156 #define KERNEL_USE_EVENTFLAG            (1)
00157
00158 #endif
```

15.51 message.cpp File Reference

Inter-thread communications via message passing.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "message.h"
#include "threadport.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** MESSAGE_CPP

15.51.1 Detailed Description

Inter-thread communications via message passing.

Definition in file message.cpp.

15.52 message.cpp

```
00001 /*===========================================================================
00002     _____       _____       _____       _____
00003  ___|    _|__  __|_    |__  __|__    |__  __| __  |__  ______
00004 |    \  /  | ||    \       ||     |     ||  |/ /     ||___   |
00005 |     \/   | ||     \      ||     \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
```

```
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "message.h"
00026 #include "threadport.h"
00027 #include "kernel_debug.h"
00028
00029 //---------------------------------------------------------------------------
00030 #if defined __FILE_ID__
00031     #undef __FILE_ID__
00032 #endif
00033 #define __FILE_ID__     MESSAGE_CPP
00034
00035
00036 #if KERNEL_USE_MESSAGE
00037
00038 #if KERNEL_USE_TIMEOUTS
00039     #include "timerlist.h"
00040 #endif
00041
00042 Message GlobalMessagePool::m_aclMessagePool[8];
00043 DoubleLinkList GlobalMessagePool::m_clList;
00044
00045 //---------------------------------------------------------------------------
00046 void GlobalMessagePool::Init()
00047 {
00048     K_UCHAR i;
00049     GlobalMessagePool::m_clList.Init();
00050     for (i = 0; i < GLOBAL_MESSAGE_POOL_SIZE; i++)
00051     {
00052         GlobalMessagePool::m_aclMessagePool[i].Init();
00053         GlobalMessagePool::m_clList.Add(&(GlobalMessagePool::m_aclMessagePool[i]));
00054     }
00055 }
00056
00057 //---------------------------------------------------------------------------
00058 void GlobalMessagePool::Push( Message *pclMessage_ )
00059 {
00060     KERNEL_ASSERT( pclMessage_ );
00061
00062     CS_ENTER();
00063
00064     GlobalMessagePool::m_clList.Add(pclMessage_);
00065
00066     CS_EXIT();
00067 }
00068
00069 //---------------------------------------------------------------------------
00070 Message *GlobalMessagePool::Pop()
00071 {
00072     Message *pclRet;
00073     CS_ENTER();
00074
00075     pclRet = static_cast<Message*>( GlobalMessagePool::m_clList.GetHead() );
00076     if (0 != pclRet)
00077     {
00078         GlobalMessagePool::m_clList.Remove( static_cast<LinkListNode*>( pclRet ) );
00079     }
00080
00081     CS_EXIT();
00082     return pclRet;
00083 }
00084
00085 //---------------------------------------------------------------------------
00086 void MessageQueue::Init()
00087 {
00088     m_clSemaphore.Init(0, GLOBAL_MESSAGE_POOL_SIZE);
00089 }
00090
00091 //---------------------------------------------------------------------------
00092 Message *MessageQueue::Receive()
00093 {
00094 #if KERNEL_USE_TIMEOUTS
00095     return Receive_i(0);
00096 #else
00097     return Receive_i();
00098 #endif
00099 }
00100
00101 //---------------------------------------------------------------------------
00102 #if KERNEL_USE_TIMEOUTS
00103 Message *MessageQueue::Receive( K_ULONG ulTimeWaitMS_)
00104 {
00105     return Receive_i( ulTimeWaitMS_ );
00106 }
00107 #endif
00108
```

```
00109 //---------------------------------------------------------------------------
00110 #if KERNEL_USE_TIMEOUTS
00111 Message *MessageQueue::Receive_i( K_ULONG ulTimeWaitMS_ )
00112 #else
00113 Message *MessageQueue::Receive_i( void )
00114 #endif
00115 {
00116     Message *pclRet;
00117 
00118     // Block the current thread on the counting semaphore
00119 #if KERNEL_USE_TIMEOUTS
00120     if (!m_clSemaphore.Pend(ulTimeWaitMS_))
00121     {
00122         return NULL;
00123     }
00124 #else
00125     m_clSemaphore.Pend();
00126 #endif
00127 
00128     CS_ENTER();
00129 
00130     // Pop the head of the message queue and return it
00131     pclRet = static_cast<Message*>( m_clLinkList.GetHead() );
00132     m_clLinkList.Remove(static_cast<Message*>(pclRet));
00133 
00134     CS_EXIT();
00135 
00136     return pclRet;
00137 }
00138 
00139 //---------------------------------------------------------------------------
00140 void MessageQueue::Send( Message *pclSrc_ )
00141 {
00142     KERNEL_ASSERT( pclSrc_ );
00143 
00144     CS_ENTER();
00145 
00146     // Add the message to the head of the linked list
00147     m_clLinkList.Add( pclSrc_ );
00148 
00149     // Post the semaphore, waking the blocking thread for the queue.
00150     m_clSemaphore.Post();
00151 
00152     CS_EXIT();
00153 }
00154 
00155 //---------------------------------------------------------------------------
00156 K_USHORT MessageQueue::GetCount()
00157 {
00158     return m_clSemaphore.GetCount();
00159 }
00160 #endif //KERNEL_USE_MESSAGE
```

15.53 message.h File Reference

Inter-thread communication via message-passing.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
#include "ksemaphore.h"
```

Classes

- class Message

  Class to provide message-based IPC services in the kernel.

- class GlobalMessagePool

  Implements a list of message objects shared between all threads.

- class MessageQueue

  List of messages, used as the channel for sending and receiving messages between threads.

15.53.1 Detailed Description

Inter-thread communication via message-passing. Embedded systems guru Jack Ganssle once said that without a robust form of interprocess communications (IPC), an RTOS is just a toy. Mark3 implements a form of IPC to provide safe and flexible messaging between threads.

Using kernel-managed IPC offers significant benefits over other forms of data sharing (i.e. Global variables) in that it avoids synchronization issues and race conditions common to the practice. Using IPC also enforces a more disciplined coding style that keeps threads decoupled from one another and minimizes global data, preventing careless and hard-to-debug errors.

15.53.2 Using Messages, Queues, and the Global Message Pool

```
// Declare a message queue shared between two threads
MessageQueue my_queue;

int main()
{
    ...
    // Initialize the message queue
    my_queue.init();
    ...
}

void Thread1()
{
    // Example TX thread - sends a message every 10ms
    while(1)
    {
        // Grab a message from the global message pool
        Message *tx_message = GlobalMessagePool::Pop();

        // Set the message data/parameters
        tx_message->SetCode( 1234 );
        tx_message->SetData( NULL );

        // Send the message on the queue.
        my_queue.Send( tx_message );
        Thread::Sleep(10);
    }
}

void Thread2()
{
    while()
    {
        // Blocking receive - wait until we have messages to process
        Message *rx_message = my_queue.Recv();

        // Do something with the message data...

        // Return back into the pool when done
        GlobalMessagePool::Push(rx_message);
    }
}
```

Definition in file message.h.

15.54 message.h

```
00001 /*=========================================================================
00002       _____          _____          _____          _____
00003  ____|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \   /   | ||     \       ||      |     ||   |/ /      ||___    |
00005 |     \/    | ||      \      ||      \     ||      \      ||___    |
00006 |__/\__/|__|_||__|\__\   __||__|\__\   __||__|\__\   __||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========================================================================*/
00080 #ifndef __MESSAGE_H__
00081 #define __MESSAGE_H__
```

```
00082
00083 #include "kerneltypes.h"
00084 #include "mark3cfg.h"
00085
00086 #include "ll.h"
00087 #include "ksemaphore.h"
00088
00089 #if KERNEL_USE_MESSAGE
00090
00091 #if KERNEL_USE_TIMEOUTS
00092     #include "timerlist.h"
00093 #endif
00094
00095 //---------------------------------------------------------------------------
00099 class Message : public LinkListNode
00100 {
00101 public:
00107     void Init() { ClearNode(); m_pvData = NULL; m_usCode = 0; }
00108
00116     void SetData( void *pvData_ ) { m_pvData = pvData_; }
00117
00125     void *GetData() { return m_pvData; }
00126
00134     void SetCode( K_USHORT usCode_ ) { m_usCode = usCode_; }
00135
00143     K_USHORT GetCode() { return m_usCode; }
00144 private:
00145
00147     void *m_pvData;
00148
00150     K_USHORT m_usCode;
00151 };
00152
00153 //---------------------------------------------------------------------------
00157 class GlobalMessagePool
00158 {
00159 public:
00165     static void Init();
00166
00176     static void Push( Message *pclMessage_ );
00177
00186     static Message *Pop();
00187
00188 private:
00190     static Message m_aclMessagePool[
      GLOBAL_MESSAGE_POOL_SIZE];
00191
00193     static DoubleLinkList m_clList;
00194 };
00195
00196 //---------------------------------------------------------------------------
00201 class MessageQueue
00202 {
00203 public:
00209     void Init();
00210
00219     Message *Receive();
00220
00221 #if KERNEL_USE_TIMEOUTS
00222
00236     Message *Receive( K_ULONG ulTimeWaitMS_ );
00237 #endif
00238
00247     void Send( Message *pclSrc_ );
00248
00249
00257     K_USHORT GetCount();
00258 private:
00259
00260 #if KERNEL_USE_TIMEOUTS
00261     Message *Receive_i( K_ULONG ulTimeWaitMS_ );
00262 #else
00263     Message *Receive_i( void );
00264 #endif
00265
00267     Semaphore m_clSemaphore;
00268
00270     DoubleLinkList m_clLinkList;
00271 };
00272
00273 #endif //KERNEL_USE_MESSAGE
00274
00275 #endif
```

15.55 mutex.cpp File Reference

Mutual-exclusion object.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
#include "mutex.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** MUTEX_CPP

15.55.1 Detailed Description

Mutual-exclusion object.

Definition in file mutex.cpp.

15.56 mutex.cpp

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00020 #include "kerneltypes.h"
00021 #include "mark3cfg.h"
00022
00023 #include "blocking.h"
00024 #include "mutex.h"
00025 #include "kernel_debug.h"
00026 //---------------------------------------------------------------------------
00027 #if defined __FILE_ID__
00028     #undef __FILE_ID__
00029 #endif
00030 #define __FILE_ID__     MUTEX_CPP
00031
00032
00033 #if KERNEL_USE_MUTEX
00034
00035 #if KERNEL_USE_TIMEOUTS
00036
00037 //---------------------------------------------------------------------------
00038 void TimedMutex_Calback(Thread *pclOwner_, void *pvData_)
00039 {
00040     Mutex *pclMutex = static_cast<Mutex*>(pvData_);
00041
00042     // Indicate that the semaphore has expired on the thread
00043     pclOwner_->SetExpired(true);
00044
00045     // Wake up the thread that was blocked on this semaphore.
00046     pclMutex->WakeMe(pclOwner_);
00047
00048     if (pclOwner_->GetPriority() > Scheduler::GetCurrentThread()->
      GetPriority())
00049     {
00050         Thread::Yield();
00051     }
00052 }
00053
00054 //---------------------------------------------------------------------------
```

```
00055 void Mutex::WakeMe(Thread *pclOwner_)
00056 {
00057     // Remove from the semaphore waitlist and back to its ready list.
00058     UnBlock(pclOwner_);
00059 }
00060
00061 #endif
00062
00063 //---------------------------------------------------------------------------
00064 K_UCHAR Mutex::WakeNext()
00065 {
00066     Thread *pclChosenOne = NULL;
00067
00068     // Get the highest priority waiter thread
00069     pclChosenOne = m_clBlockList.HighestWaiter();
00070
00071     // Unblock the thread
00072     UnBlock(pclChosenOne);
00073
00074     // The chosen one now owns the mutex
00075     m_pclOwner = pclChosenOne;
00076
00077     // Signal a context switch if it's a greater than or equal to the current priority
00078     if (pclChosenOne->GetPriority() >= Scheduler::GetCurrentThread()
      ->GetPriority())
00079     {
00080         return 1;
00081     }
00082     return 0;
00083 }
00084
00085 //---------------------------------------------------------------------------
00086 void Mutex::Init()
00087 {
00088     // Reset the data in the mutex
00089     m_bReady = 1;              // The mutex is free.
00090     m_ucMaxPri = 0;            // Set the maximum priority inheritence state
00091     m_pclOwner = NULL;         // Clear the mutex owner
00092     m_ucRecurse = 0;           // Reset recurse count
00093 }
00094
00095 //---------------------------------------------------------------------------
00096 #if KERNEL_USE_TIMEOUTS
00097 bool Mutex::Claim_i(K_ULONG ulWaitTimeMS_)
00098 #else
00099 void Mutex::Claim_i(void)
00100 #endif
00101 {
00102     KERNEL_TRACE_1( STR_MUTEX_CLAIM_1, (K_USHORT)g_pstCurrent->GetID() );
00103
00104 #if KERNEL_USE_TIMEOUTS
00105     Timer clTimer;
00106     bool bUseTimer = false;
00107 #endif
00108
00109     // Disable the scheduler while claiming the mutex - we're dealing with all
00110     // sorts of private thread data, can't have a thread switch while messing
00111     // with internal data structures.
00112     Scheduler::SetScheduler(0);
00113
00114     // Check to see if the mutex is claimed or not
00115     if (m_bReady != 0)
00116     {
00117         // Mutex isn't claimed, claim it.
00118         m_bReady = 0;
00119         m_ucRecurse = 0;
00120         m_ucMaxPri = g_pstCurrent->GetPriority();
00121         m_pclOwner = g_pstCurrent;
00122
00123         Scheduler::SetScheduler(1);
00124
00125 #if KERNEL_USE_TIMEOUTS
00126         return true;
00127 #else
00128         return;
00129 #endif
00130     }
00131
00132     // If the mutex is already claimed, check to see if this is the owner thread,
00133     // since we allow the mutex to be claimed recursively.
00134     if (g_pstCurrent == m_pclOwner)
00135     {
00136         // Ensure that we haven't exceeded the maximum recursive-lock count
00137         KERNEL_ASSERT( (m_ucRecurse < 255) );
00138         m_ucRecurse++;
00139
00140         // Increment the lock count and bail
```

```
00141         Scheduler::SetScheduler(1);
00142 #if KERNEL_USE_TIMEOUTS
00143         return true;
00144 #else
00145         return;
00146 #endif
00147     }
00148 
00149     // The mutex is claimed already - we have to block now.  Move the
00150     // current thread to the list of threads waiting on the mutex.
00151 #if KERNEL_USE_TIMEOUTS
00152     if (ulWaitTimeMS_)
00153     {
00154         g_pstCurrent->SetExpired(false);
00155         clTimer.Init();
00156         clTimer.Start(0, ulWaitTimeMS_, (TimerCallback_t)TimedMutex_Calback, (void*)this);
00157         bUseTimer = true;
00158     }
00159 #endif
00160     Block(g_pstCurrent);
00161 
00162     // Check if priority inheritence is necessary.  We do this in order
00163     // to ensure that we don't end up with priority inversions in case
00164     // multiple threads are waiting on the same resource.
00165     if(m_ucMaxPri <= g_pstCurrent->GetPriority())
00166     {
00167         m_ucMaxPri = g_pstCurrent->GetPriority();
00168 
00169         Thread *pclTemp = static_cast<Thread*>(m_clBlockList.GetHead());
00170         while(pclTemp)
00171         {
00172             pclTemp->InheritPriority(m_ucMaxPri);
00173             if(pclTemp == static_cast<Thread*>(m_clBlockList.GetTail()) )
00174             {
00175                 break;
00176             }
00177             pclTemp = static_cast<Thread*>(pclTemp->GetNext());
00178         }
00179         m_pclOwner->InheritPriority(m_ucMaxPri);
00180     }
00181 
00182     // Done with thread data -reenable the scheduler
00183     Scheduler::SetScheduler(1);
00184 
00185     // Switch threads if this thread acquired the mutex
00186     Thread::Yield();
00187 
00188 #if KERNEL_USE_TIMEOUTS
00189     if (bUseTimer)
00190     {
00191         clTimer.Stop();
00192         return (g_pstCurrent->GetExpired() == 0);
00193     }
00194     return true;
00195 #endif
00196 }
00197 
00198 //---------------------------------------------------------------------------
00199 void Mutex::Claim(void)
00200 {
00201 #if KERNEL_USE_TIMEOUTS
00202     Claim_i(0);
00203 #else
00204     Claim_i();
00205 #endif
00206 }
00207 
00208 //---------------------------------------------------------------------------
00209 #if KERNEL_USE_TIMEOUTS
00210 bool Mutex::Claim(K_ULONG ulWaitTimeMS_)
00211 {
00212     return Claim_i(ulWaitTimeMS_);
00213 }
00214 #endif
00215 
00216 //---------------------------------------------------------------------------
00217 void Mutex::Release()
00218 {
00219     KERNEL_TRACE_1( STR_MUTEX_RELEASE_1, (K_USHORT)g_pstCurrent->GetID() );
00220 
00221     bool bSchedule = 0;
00222 
00223     // Disable the scheduler while we deal with internal data structures.
00224     Scheduler::SetScheduler(0);
00225 
00226     // This thread had better be the one that owns the mutex currently...
00227     KERNEL_ASSERT( (g_pstCurrent == m_pclOwner) );
```

```
00228
00229     // If the owner had claimed the lock multiple times, decrease the lock
00230     // count and return immediately.
00231     if (m_ucRecurse)
00232     {
00233         m_ucRecurse--;
00234         Scheduler::SetScheduler(1);
00235         return;
00236     }
00237
00238     // Restore the thread's original priority
00239     if (g_pstCurrent->GetCurPriority() != g_pstCurrent->
      GetPriority())
00240     {
00241         g_pstCurrent->SetPriority(g_pstCurrent->GetPriority());
00242
00243         // In this case, we want to reschedule
00244         bSchedule = 1;
00245     }
00246
00247     // No threads are waiting on this semaphore?
00248     if (m_clBlockList.GetHead() == NULL)
00249     {
00250         // Re-initialize the mutex to its default values
00251         m_bReady = 1;
00252         m_ucMaxPri = 0;
00253         m_pclOwner = NULL;
00254     }
00255     else
00256     {
00257         // Wake the highest priority Thread pending on the mutex
00258         if(WakeNext())
00259         {
00260             // Switch threads if it's higher or equal priority than the current thread
00261             bSchedule = 1;
00262         }
00263     }
00264
00265     // Must enable the scheduler again in order to switch threads.
00266     Scheduler::SetScheduler(1);
00267     if(bSchedule)
00268     {
00269         // Switch threads if a higher-priority thread was woken
00270         Thread::Yield();
00271     }
00272 }
00273
00274 #endif //KERNEL_USE_MUTEX
```

15.57 mutex.h File Reference

Mutual exclusion class declaration.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
```

Classes

- class Mutex

  Mutual-exclusion locks, based on BlockingObject.

15.57.1 Detailed Description

Mutual exclusion class declaration. Resource locks are implemented using mutual exclusion semaphores (Mutex_t). Protected blocks can be placed around any resource that may only be accessed by one thread at a time. If additional threads attempt to access the protected resource, they will be placed in a wait queue until the resource becomes available. When the resource becomes available, the thread with the highest original priority claims the resource and is activated. Priority inheritance is included in the implementation to prevent priority inversion. Always

ensure that you claim and release your mutex objects consistently, otherwise you may end up with a deadlock scenario that's hard to debug.

15.57.2 Initializing

Initializing a mutex object by calling:

```
clMutex.Init();
```

15.57.3 Resource protection example

```
clMutex.Claim();
...
<resource protected block>
...
clMutex.Release();
```

Definition in file mutex.h.

15.58 mutex.h

```
00001 /*=============================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__   |__   ___| __  |__   ______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =============================================================================*/
00050 #ifndef __MUTEX_H_
00051 #define __MUTEX_H_
00052
00053 #include "kerneltypes.h"
00054 #include "mark3cfg.h"
00055
00056 #include "blocking.h"
00057
00058 #if KERNEL_USE_MUTEX
00059
00060 #if KERNEL_USE_TIMEOUTS
00061 #include "timerlist.h"
00062 #endif
00063
00064 //---------------------------------------------------------------------------
00068 class Mutex : public BlockingObject
00069 {
00070 public:
00077     void Init();
00078
00085     void Claim();
00086
00087 #if KERNEL_USE_TIMEOUTS
00088
00097     bool Claim(K_ULONG ulWaitTimeMS_);
00098
00111     void WakeMe( Thread *pclOwner_ );
00112
00113 #endif
00114
00121     void Release();
00122
00123 private:
00124
00130     K_UCHAR WakeNext();
00131
00132
00133 #if KERNEL_USE_TIMEOUTS
00134     bool Claim_i( K_ULONG ulWaitTimeMS_ );
00135 #else
```

```
00136     void Claim_i(void);
00137 #endif
00138
00139     K_UCHAR m_ucRecurse;
00140     K_UCHAR m_bReady;
00141     K_UCHAR m_ucMaxPri;
00142     Thread *m_pclOwner;
00143
00144 };
00145
00146 #endif //KERNEL_USE_MUTEX
00147
00148 #endif //__MUTEX_H_
00149
```

15.59 panic_codes.h File Reference

Defines the reason codes thrown when a kernel panic occurs.

Macros

- #define **PANIC_ASSERT_FAILED** (1)
- #define **PANIC_LIST_UNLINK_FAILED** (2)
- #define **PANIC_STACK_SLACK_VIOLATED** (3)

15.59.1 Detailed Description

Defines the reason codes thrown when a kernel panic occurs.

Definition in file panic_codes.h.

15.60 panic_codes.h

```
00001 /*=============================================================================
00002     _____       _____       _____       _____
00003  ___|    _|__  __|_   |__  __|__   |__  __| __  |__  ______
00004 |    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
00005 |     \/   | ||     \     ||     \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007    |_____|     |_____|     |_____|     |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00020 #ifndef __PANIC_CODES_H
00021 #define __PANIC_CODES_H
00022
00023 #define PANIC_ASSERT_FAILED         (1)
00024 #define PANIC_LIST_UNLINK_FAILED    (2)
00025 #define PANIC_STACK_SLACK_VIOLATED  (3)
00026
00027 #endif // __PANIC_CODES_H
00028
```

15.61 profile.cpp File Reference

Code profiling utilities.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "profile.h"
#include "kprofile.h"
#include "threadport.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** PROFILE_CPP

15.61.1 Detailed Description

Code profiling utilities.

Definition in file profile.cpp.

15.62 profile.cpp

```
00001 /*=========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__    |__   ___| __   |__  ______
00004 |    \  /   | ||    \       ||      |     ||  |/ /     ||___   |
00005 |     \/    | ||     \      ||      \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023 #include "profile.h"
00024 #include "kprofile.h"
00025 #include "threadport.h"
00026 #include "kernel_debug.h"
00027 //---------------------------------------------------------------------------
00028 #if defined __FILE_ID__
00029     #undef __FILE_ID__
00030 #endif
00031 #define __FILE_ID__     PROFILE_CPP
00032
00033
00034 #if KERNEL_USE_PROFILER
00035
00036 //---------------------------------------------------------------------------
00037 void ProfileTimer::Init()
00038 {
00039     m_ulCumulative = 0;
00040     m_ulCurrentIteration = 0;
00041     m_usIterations = 0;
00042     m_bActive = 0;
00043 }
00044
00045 //---------------------------------------------------------------------------
00046 void ProfileTimer::Start()
00047 {
00048     if (!m_bActive)
00049     {
00050         CS_ENTER();
00051         m_ulCurrentIteration = 0;
00052         m_ulInitialEpoch = Profiler::GetEpoch();
00053         m_usInitial = Profiler::Read();
00054         CS_EXIT();
00055         m_bActive = 1;
00056     }
00057 }
00058
00059 //---------------------------------------------------------------------------
00060 void ProfileTimer::Stop()
```

```
00061 {
00062     if (m_bActive)
00063     {
00064         K_USHORT usFinal;
00065         K_ULONG ulEpoch;
00066         CS_ENTER();
00067         usFinal = Profiler::Read();
00068         ulEpoch = Profiler::GetEpoch();
00069         // Compute total for current iteration...
00070         m_ulCurrentIteration = ComputeCurrentTicks(usFinal, ulEpoch);
00071         m_ulCumulative += m_ulCurrentIteration;
00072         m_usIterations++;
00073         CS_EXIT();
00074         m_bActive = 0;
00075     }
00076 }
00077
00078 //---------------------------------------------------------------------------
00079 K_ULONG ProfileTimer::GetAverage()
00080 {
00081     if (m_usIterations)
00082     {
00083         return m_ulCumulative / (K_ULONG)m_usIterations;
00084     }
00085     return 0;
00086 }
00087
00088 //---------------------------------------------------------------------------
00089 K_ULONG ProfileTimer::GetCurrent()
00090 {
00091
00092     if (m_bActive)
00093     {
00094         K_USHORT usCurrent;
00095         K_ULONG ulEpoch;
00096         CS_ENTER();
00097         usCurrent = Profiler::Read();
00098         ulEpoch = Profiler::GetEpoch();
00099         CS_EXIT();
00100         return ComputeCurrentTicks(usCurrent, ulEpoch);
00101     }
00102     return m_ulCurrentIteration;
00103 }
00104
00105 //---------------------------------------------------------------------------
00106 K_ULONG ProfileTimer::ComputeCurrentTicks(K_USHORT usCurrent_, K_ULONG ulEpoch_)
00107 {
00108     K_ULONG ulTotal;
00109     K_ULONG ulOverflows;
00110
00111     ulOverflows = ulEpoch_ - m_ulInitialEpoch;
00112
00113     // More than one overflow...
00114     if (ulOverflows > 1)
00115     {
00116         ulTotal = ((K_ULONG)(ulOverflows-1) * TICKS_PER_OVERFLOW)
00117                 + (K_ULONG)(TICKS_PER_OVERFLOW - m_usInitial) +
00118                 (K_ULONG)usCurrent_;
00119     }
00120     // Only one overflow, or one overflow that has yet to be processed
00121     else if (ulOverflows || (usCurrent_ < m_usInitial))
00122     {
00123         ulTotal = (K_ULONG)(TICKS_PER_OVERFLOW - m_usInitial) +
00124                 (K_ULONG)usCurrent_;
00125     }
00126     // No overflows, none pending.
00127     else
00128     {
00129         ulTotal = (K_ULONG)(usCurrent_ - m_usInitial);
00130     }
00131
00132     return ulTotal;
00133 }
00134
00135 #endif
```

15.63 profile.h File Reference

High-precision profiling timers.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
```

15.63.1 Detailed Description

High-precision profiling timers. Enables the profiling and instrumentation of performance-critical code. Multiple timers can be used simultaneously to enable system-wide performance metrics to be computed in a lightweight manner.

Usage:

```
ProfileTimer clMyTimer;
int i;

clMyTimer.Init();

// Profile the same block of code ten times
for (i = 0; i < 10; i++)
{
    clMyTimer.Start();
    ...
    //Block of code to profile
    ...
    clMyTimer.Stop();
}

// Get the average execution time of all iterations
ulAverageTimer = clMyTimer.GetAverage();

// Get the execution time from the last iteration
ulLastTimer = clMyTimer.GetCurrent();
```

Definition in file profile.h.

15.64 profile.h

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|     |__   ___|_    |__   ___|__   |__   ___| __  |__  ______
00004 |    \   /  | ||    \       ||      |     ||   |/ /     ||___    |
00005 |     \/    | ||     \      ||      \     ||     \      ||___    |
00006 |__/\__/|__|_||__|\__\  ___||__|\__\  ___||__|\__\  ___||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00053 #ifndef __PROFILE_H__
00054 #define __PROFILE_H__
00055
00056 #include "kerneltypes.h"
00057 #include "mark3cfg.h"
00058 #include "ll.h"
00059
00060 #if KERNEL_USE_PROFILER
00061
00069 class ProfileTimer
00070 {
00071
00072 public:
00079     void Init();
00080
00087     void Start();
00088
00095     void Stop();
00096
00104     K_ULONG GetAverage();
00105
00114     K_ULONG GetCurrent();
00115
00116 private:
```

```
00117
00126     K_ULONG ComputeCurrentTicks(K_USHORT usCount_, K_ULONG ulEpoch_);
00127
00128     K_ULONG m_ulCumulative;
00129     K_ULONG m_ulCurrentIteration;
00130     K_USHORT m_usInitial;
00131     K_ULONG m_ulInitialEpoch;
00132     K_USHORT m_usIterations;
00133     K_UCHAR m_bActive;
00134 };
00135
00136 #endif // KERNEL_USE_PROFILE
00137
00138 #endif
```

15.65 quantum.cpp File Reference

Thread Quantum Implementation for Round-Robin Scheduling.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "timerlist.h"
#include "quantum.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** QUANTUM_CPP

Functions

- static void **QuantumCallback** (Thread *pclThread_, void *pvData_)

Variables

- static volatile K_BOOL **bAddQuantumTimer**

15.65.1 Detailed Description

Thread Quantum Implementation for Round-Robin Scheduling.

Definition in file quantum.cpp.

15.66 quantum.cpp

```
00001 /*===========================================================================
00002       _____          _____          _____          _____
00003  ____|    _|__   ___|_    |___   ___|___    |___  ___| ___  |___  ______
00004 |    \  /  | ||    \       ||     |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||     \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  ___||__|\__\  ___||__|\__\  ___||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
```

```
00024
00025 #include "thread.h"
00026 #include "timerlist.h"
00027 #include "quantum.h"
00028 #include "kernel_debug.h"
00029 //---------------------------------------------------------------------------
00030 #if defined __FILE_ID__
00031     #undef __FILE_ID__
00032 #endif
00033 #define __FILE_ID__     QUANTUM_CPP
00034
00035 #if KERNEL_USE_QUANTUM
00036
00037 //---------------------------------------------------------------------------
00038 static volatile K_BOOL bAddQuantumTimer;    // Indicates that a timer add is pending
00039
00040 //---------------------------------------------------------------------------
00041 Timer Quantum::m_clQuantumTimer;   // The global timernodelist_t object
00042 K_UCHAR Quantum::m_bActive;
00043 K_UCHAR Quantum::m_bInTimer;
00044 //---------------------------------------------------------------------------
00045 static void QuantumCallback(Thread *pclThread_, void *pvData_)
00046 {
00047     // Validate thread pointer, check that source/destination match (it's
00048     // in its real priority list).  Also check that this thread was part of
00049     // the highest-running priority level.
00050     if (pclThread_->GetPriority() >= Scheduler::GetCurrentThread()->
      GetPriority())
00051     {
00052         if (pclThread_->GetCurrent()->GetHead() != pclThread_->
      GetCurrent()->GetTail() )
00053         {
00054             bAddQuantumTimer = true;
00055             pclThread_->GetCurrent()->PivotForward();
00056         }
00057     }
00058 }
00059
00060 //---------------------------------------------------------------------------
00061 void Quantum::SetTimer(Thread *pclThread_)
00062 {
00063     m_clQuantumTimer.SetIntervalMSeconds(pclThread_->
      GetQuantum());
00064     m_clQuantumTimer.SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00065     m_clQuantumTimer.SetData(NULL);
00066     m_clQuantumTimer.SetCallback((TimerCallback_t)QuantumCallback);
00067     m_clQuantumTimer.SetOwner(pclThread_);
00068 }
00069
00070 //---------------------------------------------------------------------------
00071 void Quantum::AddThread(Thread *pclThread_)
00072 {
00073     if (m_bActive)
00074     {
00075         return;
00076     }
00077
00078     // If this is called from the timer callback, queue a timer add...
00079     if (m_bInTimer)
00080     {
00081         bAddQuantumTimer = true;
00082         return;
00083     }
00084
00085     // If this isn't the only thread in the list.
00086     if ( pclThread_->GetCurrent()->GetHead() !=
00087            pclThread_->GetCurrent()->GetTail() )
00088     {
00089
00090         Quantum::SetTimer(pclThread_);
00091         TimerScheduler::Add(&m_clQuantumTimer);
00092         m_bActive = 1;
00093     }
00094 }
00095
00096 //---------------------------------------------------------------------------
00097 void Quantum::RemoveThread(void)
00098 {
00099     if (!m_bActive)
00100     {
00101         return;
00102     }
00103
00104     // Cancel the current timer
00105     TimerScheduler::Remove(&m_clQuantumTimer);
00106     m_bActive = 0;
00107 }
```

```
00108
00109 //---------------------------------------------------------------------------
00110 void Quantum::UpdateTimer(void)
00111 {
00112     // If we have to re-add the quantum timer (more than 2 threads at the
00113     // high-priority level...)
00114     if (bAddQuantumTimer)
00115     {
00116         // Trigger a thread yield - this will also re-schedule the
00117         // thread *and* reset the round-robin scheduler.
00118         Thread::Yield();
00119         bAddQuantumTimer = false;
00120     }
00121 }
00122
00123 #endif //KERNEL_USE_QUANTUM
```

15.67 quantum.h File Reference

Thread Quantum declarations for Round-Robin Scheduling.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "timerlist.h"
```

Classes

- class Quantum

  Static-class used to implement Thread quantum functionality, which is a key part of round-robin scheduling.

15.67.1 Detailed Description

Thread Quantum declarations for Round-Robin Scheduling.

Definition in file quantum.h.

15.68 quantum.h

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   __|_    |__   __|__   |__   __| __  |__   ______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #ifndef __KQUANTUM_H__
00023 #define __KQUANTUM_H__
00024
00025 #include "kerneltypes.h"
00026 #include "mark3cfg.h"
00027
00028 #include "thread.h"
00029 #include "timerlist.h"
00030
00031 #if KERNEL_USE_QUANTUM
00032 class Timer;
00033
00039 class Quantum
00040 {
00041 public:
00050     static void UpdateTimer();
```

```
00051
00058     static void AddThread( Thread *pclThread_ );
00059
00065     static void RemoveThread();
00066
00075     static void SetInTimer(void)   { m_bInTimer = true; }
00076
00082     static void ClearInTimer(void) { m_bInTimer = false; }
00083
00084 private:
00096     static void SetTimer( Thread *pclThread_ );
00097
00098     static Timer m_clQuantumTimer;
00099     static K_UCHAR m_bActive;
00100     static K_UCHAR m_bInTimer;
00101 };
00102
00103 #endif //KERNEL_USE_QUANTUM
00104
00105 #endif
```

15.69 scheduler.cpp File Reference

Strict-Priority + Round-Robin thread scheduler implementation.

```
#include "kerneltypes.h"
#include "ll.h"
#include "scheduler.h"
#include "thread.h"
#include "threadport.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** SCHEDULER_CPP

Variables

- Thread * **g_pstNext**
- Thread * **g_pstCurrent**
- K_UCHAR **g_ucFlag**
- static const K_UCHAR **aucCLZ** [16] ={255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3}

15.69.1 Detailed Description

Strict-Priority + Round-Robin thread scheduler implementation.

Definition in file scheduler.cpp.

15.70 scheduler.cpp

```
00001 /*=============================================================================
00002     _____     _____     _____     _____
00003  ___|    _|___  __|_    |___  ___|___  ___| __  |__  ______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|     |_____|     |_____|     |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
```

```
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #include "kerneltypes.h"
00023 #include "ll.h"
00024 #include "scheduler.h"
00025 #include "thread.h"
00026 #include "threadport.h"
00027 #include "kernel_debug.h"
00028 //---------------------------------------------------------------------------
00029 #if defined __FILE_ID__
00030     #undef __FILE_ID__
00031 #endif
00032 #define __FILE_ID__     SCHEDULER_CPP
00033
00034 //---------------------------------------------------------------------------
00035 Thread *g_pstNext;
00036 Thread *g_pstCurrent;
00037
00038 //---------------------------------------------------------------------------
00039 K_BOOL Scheduler::m_bEnabled;
00040 K_BOOL Scheduler::m_bQueuedSchedule;
00041
00042 ThreadList Scheduler::m_clStopList;
00043 ThreadList Scheduler::m_aclPriorities[NUM_PRIORITIES];
00044 K_UCHAR Scheduler::m_ucPriFlag;
00045
00046 K_UCHAR g_ucFlag;
00047 //---------------------------------------------------------------------------
00048 static const K_UCHAR aucCLZ[16] ={255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3};
00049
00050 //---------------------------------------------------------------------------
00051 void Scheduler::Init()
00052 {
00053     m_ucPriFlag = 0;
00054     for (int i = 0; i < NUM_PRIORITIES; i++)
00055     {
00056         m_aclPriorities[i].SetPriority(i);
00057         m_aclPriorities[i].SetFlagPointer(&
      m_ucPriFlag);
00058     }
00059     g_ucFlag = m_ucPriFlag;
00060     m_bQueuedSchedule = false;
00061 }
00062
00063 //---------------------------------------------------------------------------
00064 void Scheduler::Schedule()
00065 {
00066     K_UCHAR ucPri = 0;
00067
00068     // Figure out what priority level has ready tasks (8 priorities max)
00069     ucPri = aucCLZ[m_ucPriFlag >> 4 ];
00070     if (ucPri == 0xFF) { ucPri = aucCLZ[m_ucPriFlag & 0x0F]; }
00071     else { ucPri += 4; }
00072
00073     // Get the thread node at this priority.
00074     g_pstNext = (Thread*)( m_aclPriorities[ucPri].GetHead() );
00075     g_ucFlag = m_ucPriFlag;
00076
00077     KERNEL_TRACE_1( STR_SCHEDULE_1, (K_USHORT)g_pstNext->GetID() );
00078 }
00079
00080 //---------------------------------------------------------------------------
00081 void Scheduler::Add(Thread *pclThread_)
00082 {
00083     m_aclPriorities[pclThread_->GetPriority()].Add(pclThread_);
00084     g_ucFlag = m_ucPriFlag;
00085 }
00086
00087 //---------------------------------------------------------------------------
00088 void Scheduler::Remove(Thread *pclThread_)
00089 {
00090     m_aclPriorities[pclThread_->GetPriority()].Remove(pclThread_);
00091     g_ucFlag = m_ucPriFlag;
00092 }
00093
00094 //---------------------------------------------------------------------------
00095 K_BOOL Scheduler::SetScheduler(K_BOOL bEnable_)
00096 {
00097     K_BOOL bRet ;
00098     CS_ENTER();
00099     bRet = m_bEnabled;
00100     m_bEnabled = bEnable_;
00101     // If there was a queued scheduler evevent, dequeue and trigger an
00102     // immediate Yield
00103     if (m_bEnabled && m_bQueuedSchedule)
00104     {
00105         m_bQueuedSchedule = false;
```

```
00106         Thread::Yield();
00107     }
00108     CS_EXIT();
00109     return bRet;
00110 }
```

15.71 scheduler.h File Reference

Thread scheduler function declarations.

```
#include "kerneltypes.h"
#include "thread.h"
#include "threadport.h"
```

Classes

- class Scheduler

  Priority-based round-robin Thread scheduling, using ThreadLists for housekeeping.

Macros

- #define **NUM_PRIORITIES** (8)

Variables

- Thread * **g_pstNext**
- Thread * **g_pstCurrent**

15.71.1 Detailed Description

Thread scheduler function declarations. This scheduler implements a very flexible type of scheduling, which has become the defacto industry standard when it comes to real-time operating systems. This scheduling mechanism is referred to as priority round- robin.

From the name, there are two concepts involved here:

1) Priority scheduling:

Threads are each assigned a priority, and the thread with the highest priority which is ready to run gets to execute.

2) Round-robin scheduling:

Where there are multiple ready threads at the highest-priority level, each thread in that group gets to share time, ensuring that progress is made.

The scheduler uses an array of ThreadList objects to provide the necessary housekeeping required to keep track of threads at the various priorities. As s result, the scheduler contains one ThreadList per priority, with an additional list to manage the storage of threads which are in the "stopped" state (either have been stopped, or have not been started yet).

Definition in file scheduler.h.

15.72 scheduler.h

```
00001 /*=============================================================================
00002     _____       _____       _____       _____
00003  ___|    _|__  __|_    |__  __|__    |__  __| __  |__  ______
```

```
00004 |    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
00005 |     \/   | ||     \     ||     \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ============================================================================*/
00046 #ifndef __SCHEDULER_H__
00047 #define __SCHEDULER_H__
00048
00049 #include "kerneltypes.h"
00050 #include "thread.h"
00051 #include "threadport.h"
00052
00053 extern Thread *g_pstNext;
00054 extern Thread *g_pstCurrent;
00055
00056 #define NUM_PRIORITIES              (8)
00057 //---------------------------------------------------------------------------
00062 class Scheduler
00063 {
00064 public:
00070     static void Init();
00071
00079     static void Schedule();
00080
00088     static void Add(Thread *pclThread_);
00089
00098     static void Remove(Thread *pclThread_);
00099
00112     static K_BOOL SetScheduler(K_BOOL bEnable_);
00113
00119     static Thread *GetCurrentThread(){ return g_pstCurrent; }
00120
00127     static Thread *GetNextThread(){ return g_pstNext; }
00128
00137     static ThreadList *GetThreadList(K_UCHAR ucPriority_){ return &
      m_aclPriorities[ucPriority_]; }
00138
00145     static ThreadList *GetStopList(){ return &m_clStopList; }
00146
00155     static K_UCHAR IsEnabled(){ return m_bEnabled; }
00156
00157     static void QueueScheduler() { m_bQueuedSchedule = true; }
00158
00159 private:
00161     static K_BOOL m_bEnabled;
00162
00164     static K_BOOL m_bQueuedSchedule;
00165
00167     static ThreadList m_clStopList;
00168
00170     static ThreadList m_aclPriorities[NUM_PRIORITIES];
00171
00173     static K_UCHAR m_ucPriFlag;
00174 };
00175 #endif
00176
```

15.73 thread.cpp File Reference

Platform-Independent thread class Definition.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "scheduler.h"
#include "kernelswi.h"
#include "timerlist.h"
#include "ksemaphore.h"
#include "quantum.h"
#include "kernel.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** THREAD_CPP

Functions

- static void ThreadSleepCallback (Thread *pclOwner_, void *pvData_)

  This callback is used to wake up a thread once the interval has expired.

15.73.1 Detailed Description

Platform-Independent thread class Definition.

Definition in file thread.cpp.

15.74 thread.cpp

```
00001 /*=============================================================================
00002       _____          _____          _____          _____
00003  ____|    _|__   ___|_    |__   ___|__   |__   ___| __   |__   ______
00004 |    \   /   | ||     \       ||      |      ||   |/ /      ||___    |
00005 |     \/    | ||      \      ||      \      ||      \      ||___    |
00006 |__/\__/|__|_||__|\__\   ___||__|\__\   ___||__|\__\   ___||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "thread.h"
00026 #include "scheduler.h"
00027 #include "kernelswi.h"
00028 #include "timerlist.h"
00029 #include "ksemaphore.h"
00030 #include "quantum.h"
00031 #include "kernel.h"
00032 #include "kernel_debug.h"
00033
00034 //---------------------------------------------------------------------------
00035 #if defined __FILE_ID__
00036     #undef __FILE_ID__
00037 #endif
00038 #define __FILE_ID__     THREAD_CPP
00039
00040 //---------------------------------------------------------------------------
00041 void Thread::Init(  K_WORD *pwStack_,
00042                 K_USHORT usStackSize_,
00043                 K_UCHAR ucPriority_,
00044                 ThreadEntry_t pfEntryPoint_,
00045                 void *pvArg_ )
00046 {
00047     static K_UCHAR ucThreadID = 0;
00048
00049     KERNEL_ASSERT( pwStack_ );
00050     KERNEL_ASSERT( pfEntryPoint_ );
00051
00052     ClearNode();
00053
00054     m_ucThreadID = ucThreadID++;
00055
00056     KERNEL_TRACE_1( STR_STACK_SIZE_1, usStackSize_ );
00057     KERNEL_TRACE_1( STR_PRIORITY_1, (K_UCHAR)ucPriority_ );
00058     KERNEL_TRACE_1( STR_THREAD_ID_1, (K_USHORT)m_ucThreadID );
00059     KERNEL_TRACE_1( STR_ENTRYPOINT_1, (K_USHORT)pfEntryPoint_ );
00060
00061     // Initialize the thread parameters to their initial values.
00062     m_pwStack = pwStack_;
00063     m_pwStackTop = TOP_OF_STACK(pwStack_, usStackSize_);
00064
00065     m_usStackSize = usStackSize_;
00066
```

```
00067 #if KERNEL_USE_QUANTUM
00068     m_usQuantum = 4;
00069 #endif
00070 
00071     m_ucPriority = ucPriority_;
00072     m_ucCurPriority = m_ucPriority;
00073     m_pfEntryPoint = pfEntryPoint_;
00074     m_pvArg = pvArg_;
00075 
00076 #if KERNEL_USE_THREADNAME
00077     m_szName = NULL;
00078 #endif
00079 
00080     // Call CPU-specific stack initialization
00081     ThreadPort::InitStack(this);
00082 
00083     // Add to the global "stop" list.
00084     CS_ENTER();
00085     m_pclOwner = Scheduler::GetThreadList(
      m_ucPriority);
00086     m_pclCurrent = Scheduler::GetStopList();
00087     m_pclCurrent->Add(this);
00088     CS_EXIT();
00089 }
00090 
00091 //---------------------------------------------------------------------------
00092 void Thread::Start(void)
00093 {
00094     // Remove the thread from the scheduler's "stopped" list, and add it
00095     // to the scheduler's ready list at the proper priority.
00096     KERNEL_TRACE_1( STR_THREAD_START_1, (K_USHORT)m_ucThreadID );
00097 
00098     CS_ENTER();
00099     Scheduler::GetStopList()->Remove(this);
00100     Scheduler::Add(this);
00101     m_pclOwner = Scheduler::GetThreadList(
      m_ucPriority);
00102     m_pclCurrent = m_pclOwner;
00103 
00104     if (Kernel::IsStarted())
00105     {
00106         if (m_ucPriority >= Scheduler::GetCurrentThread()->
      GetCurPriority())
00107         {
00108 #if KERNEL_USE_QUANTUM
00109             // Deal with the thread Quantum
00110             Quantum::RemoveThread();
00111             Quantum::AddThread(this);
00112 #endif
00113         }
00114         if (m_ucPriority > Scheduler::GetCurrentThread()->
      GetPriority())
00115         {
00116             Thread::Yield();
00117         }
00118     }
00119     CS_EXIT();
00120 }
00121 
00122 //---------------------------------------------------------------------------
00123 void Thread::Stop()
00124 {
00125     bool bReschedule = 0;
00126 
00127     CS_ENTER();
00128 
00129     // If a thread is attempting to stop itself, ensure we call the scheduler
00130     if (this == Scheduler::GetCurrentThread())
00131     {
00132         bReschedule = true;
00133     }
00134 
00135     // Add this thread to the stop-list (removing it from active scheduling)
00136     Scheduler::Remove(this);
00137     m_pclOwner = Scheduler::GetStopList();
00138     m_pclCurrent = m_pclOwner;
00139     m_pclOwner->Add(this);
00140 
00141     CS_EXIT();
00142 
00143     if (bReschedule)
00144     {
00145         Thread::Yield();
00146     }
00147 }
00148 
00149 #if KERNEL_USE_DYNAMIC_THREADS
```

```
00150 //---------------------------------------------------------------------------
00151 void Thread::Exit()
00152 {
00153     bool bReschedule = 0;
00154 
00155     KERNEL_TRACE_1( STR_THREAD_EXIT_1, m_ucThreadID );
00156 
00157     CS_ENTER();
00158 
00159     // If this thread is the actively-running thread, make sure we run the
00160     // scheduler again.
00161     if (this == Scheduler::GetCurrentThread())
00162     {
00163         bReschedule = 1;
00164     }
00165 
00166     // Remove the thread from scheduling
00167     m_pclCurrent->Remove(this);
00168 
00169 #if KERNEL_USE_TIMERS
00170     // Just to be safe - attempt to remove the thread's timer
00171     // from the timer-scheduler (does no harm if it isn't
00172     // in the timer-list)
00173     TimerScheduler::Remove(&m_clTimer);
00174 #endif
00175 
00176     CS_EXIT();
00177 
00178     if (bReschedule)
00179     {
00180         // Choose a new "next" thread if we must
00181         Thread::Yield();
00182     }
00183 }
00184 #endif
00185 
00186 #if KERNEL_USE_SLEEP
00187 //---------------------------------------------------------------------------
00189 static void ThreadSleepCallback( Thread *pclOwner_, void *pvData_ )
00190 {
00191     Semaphore *pclSemaphore = static_cast<Semaphore*>(pvData_);
00192     // Post the semaphore, which will wake the sleeping thread.
00193     pclSemaphore->Post();
00194 }
00195 
00196 //---------------------------------------------------------------------------
00197 void Thread::Sleep(K_ULONG ulTimeMs_)
00198 {
00199     Semaphore clSemaphore;
00200     Timer *pclTimer = g_pstCurrent->GetTimer();
00201 
00202     // Create a semaphore that this thread will block on
00203     clSemaphore.Init(0, 1);
00204 
00205     // Create a one-shot timer that will call a callback that posts the
00206     // semaphore, waking our thread.
00207     pclTimer->Init();
00208     pclTimer->SetIntervalMSeconds(ulTimeMs_);
00209     pclTimer->SetCallback(ThreadSleepCallback);
00210     pclTimer->SetData((void*)&clSemaphore);
00211     pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00212 
00213     // Add the new timer to the timer scheduler, and block the thread
00214     TimerScheduler::Add(pclTimer);
00215     clSemaphore.Pend();
00216 }
00217 
00218 //---------------------------------------------------------------------------
00219 void Thread::USleep(K_ULONG ulTimeUs_)
00220 {
00221     Semaphore clSemaphore;
00222     Timer *pclTimer = g_pstCurrent->GetTimer();
00223 
00224     // Create a semaphore that this thread will block on
00225     clSemaphore.Init(0, 1);
00226 
00227     // Create a one-shot timer that will call a callback that posts the
00228     // semaphore, waking our thread.
00229     pclTimer->Init();
00230     pclTimer->SetIntervalUSeconds(ulTimeUs_);
00231     pclTimer->SetCallback(ThreadSleepCallback);
00232     pclTimer->SetData((void*)&clSemaphore);
00233     pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00234 
00235     // Add the new timer to the timer scheduler, and block the thread
00236     TimerScheduler::Add(pclTimer);
00237     clSemaphore.Pend();
```

```
00238 }
00239 #endif // KERNEL_USE_SLEEP
00240
00241 //---------------------------------------------------------------------------
00242 K_USHORT Thread::GetStackSlack()
00243 {
00244     K_USHORT usCount = 0;
00245
00246     CS_ENTER();
00247
00249     for (usCount = 0; usCount < m_usStackSize; usCount++)
00250     {
00251         if (m_pwStack[usCount] != 0xFF)
00252         {
00253             break;
00254         }
00255     }
00256
00257     CS_EXIT();
00258
00259     return usCount;
00260 }
00261
00262 //---------------------------------------------------------------------------
00263 void Thread::Yield()
00264 {
00265     CS_ENTER();
00266
00267     // Run the scheduler
00268     if (Scheduler::IsEnabled())
00269     {
00270         Scheduler::Schedule();
00271
00272         // Only switch contexts if the new task is different than the old task
00273         if (Scheduler::GetCurrentThread() !=
    Scheduler::GetNextThread())
00274         {
00275 #if KERNEL_USE_QUANTUM
00276             // new thread scheduled.  Stop current quantum timer (if it exists),
00277             // and restart it for the new thread (if required).
00278             Quantum::RemoveThread();
00279             Quantum::AddThread(g_pstNext);
00280 #endif
00281             Thread::ContextSwitchSWI();
00282         }
00283     }
00284     else
00285     {
00286         Scheduler::QueueScheduler();
00287     }
00288
00289     CS_EXIT();
00290 }
00291
00292 //---------------------------------------------------------------------------
00293 void Thread::SetPriorityBase(K_UCHAR ucPriority_)
00294 {
00295      GetCurrent()->Remove(this);
00296
00297      SetCurrent(Scheduler::GetThreadList(
    m_ucPriority));
00298
00299      GetCurrent()->Add(this);
00300 }
00301
00302 //---------------------------------------------------------------------------
00303 void Thread::SetPriority(K_UCHAR ucPriority_)
00304 {
00305     bool bSchedule = 0;
00306     CS_ENTER();
00307     // If this is the currently running thread, it's a good idea to reschedule
00308     // Or, if the new priority is a higher priority than the current thread's.
00309     if ((g_pstCurrent == this) || (ucPriority_ > g_pstCurrent->GetPriority()))
00310     {
00311         bSchedule = 1;
00312     }
00313     Scheduler::Remove(this);
00314     CS_EXIT();
00315
00316     m_ucCurPriority = ucPriority_;
00317     m_ucPriority = ucPriority_;
00318
00319     CS_ENTER();
00320     Scheduler::Add(this);
00321     CS_EXIT();
00322
00323     if (bSchedule)
```

```
00324     {
00325         if (Scheduler::IsEnabled())
00326         {
00327             CS_ENTER();
00328             Scheduler::Schedule();
00329     #if KERNEL_USE_QUANTUM
00330             // new thread scheduled.  Stop current quantum timer (if it exists),
00331             // and restart it for the new thread (if required).
00332             Quantum::RemoveThread();
00333             Quantum::AddThread(g_pstNext);
00334     #endif
00335             CS_EXIT();
00336             Thread::ContextSwitchSWI();
00337         }
00338         else
00339         {
00340             Scheduler::QueueScheduler();
00341         }
00342     }
00343 }
00344
00345 //---------------------------------------------------------------------------
00346 void Thread::InheritPriority(K_UCHAR ucPriority_)
00347 {
00348     SetOwner(Scheduler::GetThreadList(ucPriority_));
00349     m_ucCurPriority = ucPriority_;
00350 }
00351
00352 //---------------------------------------------------------------------------
00353 void Thread::ContextSwitchSWI()
00354 {
00355     // Call the context switch interrupt if the scheduler is enabled.
00356     if (Scheduler::IsEnabled() == 1)
00357     {
00358         KERNEL_TRACE_1( STR_CONTEXT_SWITCH_1, (K_USHORT)g_pstNext->GetID() );
00359         KernelSWI::Trigger();
00360     }
00361 }
00362
00363 #if KERNEL_USE_TIMERS
00364 //---------------------------------------------------------------------------
00365 Timer *Thread::GetTimer()                     { return &
      m_clTimer; }
00366 //---------------------------------------------------------------------------
00367
00368 void Thread::SetExpired( K_BOOL bExpired_ )      { m_bExpired = bExpired_; }
00369 //---------------------------------------------------------------------------
00370
00371 K_BOOL Thread::GetExpired()                      { return m_bExpired; }
00372 #endif
00373
```

15.75 thread.h File Reference

Platform independent thread class declarations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
#include "threadlist.h"
#include "scheduler.h"
#include "threadport.h"
#include "quantum.h"
```

Classes

- class Thread

  Object providing fundamental multitasking support in the kernel.

Typedefs

- typedef void(∗ ThreadEntry_t)(void ∗pvArg_)

  Function pointer type used for thread entrypoint functions.

15.75.1 Detailed Description

Platform independent thread class declarations. Threads are an atomic unit of execution, and each instance of the thread class represents an instance of a program running of the processor. The Thread is the fundmanetal user-facing object in the kernel - it is what makes multiprocessing possible from application code.

In Mark3, threads each have their own context - consisting of a stack, and all of the registers required to multiplex a processor between multiple threads.

The Thread class inherits directly from the LinkListNode class to facilitate efficient thread management using Double, or Double-Circular linked lists.

Definition in file thread.h.

15.76 thread.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|__   |__  __| __  |__  _______
00004 |    \  /  | ||    \      ||     |     ||  |/ /     ||___    |
00005 |     \/   | ||     \     ||     \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00035 #ifndef __THREAD_H__
00036 #define __THREAD_H__
00037
00038 #include "kerneltypes.h"
00039 #include "mark3cfg.h"
00040
00041 #include "ll.h"
00042 #include "threadlist.h"
00043 #include "scheduler.h"
00044 #include "threadport.h"
00045 #include "quantum.h"
00046
00047 //---------------------------------------------------------------------------
00051 typedef void (*ThreadEntry_t)(void *pvArg_);
00052
00053 //---------------------------------------------------------------------------
00057 class Thread : public LinkListNode
00058 {
00059 public:
00079     void Init(K_WORD *paucStack_,
00080               K_USHORT usStackSize_,
00081               K_UCHAR ucPriority_,
00082               ThreadEntry_t pfEntryPoint_,
00083               void *pvArg_ );
00084
00092     void Start();
00093
00094
00101     void Stop();
00102
00103 #if KERNEL_USE_THREADNAME
00104
00113     void SetName(const K_CHAR *szName_) { m_szName = szName_; }
00114
00121     const K_CHAR* GetName() { return m_szName; }
00122 #endif
00123
00132     ThreadList *GetOwner(void) { return m_pclOwner; }
00133
00141     ThreadList *GetCurrent(void) { return m_pclCurrent; }
00142
```

```
00151     K_UCHAR GetPriority(void) { return m_ucPriority; }
00152
00160     K_UCHAR GetCurPriority(void) { return m_ucCurPriority; }
00161
00162 #if KERNEL_USE_QUANTUM
00163
00170     void SetQuantum( K_USHORT usQuantum_ ) { m_usQuantum = usQuantum_; }
00171
00179     K_USHORT GetQuantum(void) { return m_usQuantum; }
00180 #endif
00181
00189     void SetCurrent( ThreadList *pclNewList_ ) {
      m_pclCurrent = pclNewList_; }
00190
00198     void SetOwner( ThreadList *pclNewList_ ) { m_pclOwner = pclNewList_; }
00199
00200
00213     void SetPriority(K_UCHAR ucPriority_);
00214
00224     void InheritPriority(K_UCHAR ucPriority_);
00225
00226 #if KERNEL_USE_DYNAMIC_THREADS
00227
00238     void Exit();
00239 #endif
00240
00241 #if KERNEL_USE_SLEEP
00242
00250     static void Sleep(K_ULONG ulTimeMs_);
00251
00260     static void USleep(K_ULONG ulTimeUs_);
00261 #endif
00262
00270     static void Yield(void);
00271
00279     void SetID( K_UCHAR ucID_ ) { m_ucThreadID = ucID_; }
00280
00288     K_UCHAR GetID() { return m_ucThreadID; }
00289
00290
00303     K_USHORT GetStackSlack();
00304
00305 #if KERNEL_USE_EVENTFLAG
00306
00313     K_USHORT GetEventFlagMask() { return m_usFlagMask; }
00314
00319     void SetEventFlagMask(K_USHORT usMask_) { m_usFlagMask = usMask_; }
00320
00326     void SetEventFlagMode(EventFlagOperation_t eMode_ ) {
      m_eFlagMode = eMode_; }
00327
00332     EventFlagOperation_t GetEventFlagMode() { return m_eFlagMode; }
00333 #endif
00334
00335 #if KERNEL_USE_TIMERS
00336
00339     Timer *GetTimer();
00340     void SetExpired( K_BOOL bExpired_ );
00341     K_BOOL GetExpired();
00342 #endif
00343
00344     friend class ThreadPort;
00345
00346 private:
00354     static void ContextSwitchSWI(void);
00355
00360     void SetPriorityBase(K_UCHAR ucPriority_);
00361
00363     K_WORD *m_pwStackTop;
00364
00366     K_WORD *m_pwStack;
00367
00369     K_USHORT m_usStackSize;
00370
00371 #if KERNEL_USE_QUANTUM
00372     K_USHORT m_usQuantum;
00374 #endif
00375
00377     K_UCHAR m_ucThreadID;
00378
00380     K_UCHAR m_ucPriority;
00381
00383     K_UCHAR m_ucCurPriority;
00384
00386     ThreadEntry_t m_pfEntryPoint;
00387
00389     void *m_pvArg;
```

```
00390
00391 #if KERNEL_USE_THREADNAME
00392     const K_CHAR *m_szName;
00394 #endif
00395
00396 #if KERNEL_USE_EVENTFLAG
00397     K_USHORT m_usFlagMask;
00399
00401     EventFlagOperation_t m_eFlagMode;
00402 #endif
00403
00404 #if KERNEL_USE_TIMERS
00405     Timer  m_clTimer;
00407     K_BOOL  m_bExpired;
00408 #endif
00409
00411     ThreadList *m_pclCurrent;
00412
00414     ThreadList *m_pclOwner;
00415 };
00416
00417 #endif
```

15.77 threadlist.cpp File Reference

Thread linked-list definitions.

```
#include "kerneltypes.h"
#include "ll.h"
#include "threadlist.h"
#include "thread.h"
#include "kernel_debug.h"
```

Macros

- #define **__FILE_ID__** THREADLIST_CPP

15.77.1 Detailed Description

Thread linked-list definitions.

Definition in file threadlist.cpp.

15.78 threadlist.cpp

```
00001 /*=============================================================================
00002       _____          _____          _____          _____
00003  ____|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \  /   | ||     \       ||       |     ||   |/ /      ||___    |
00005 |     \/    | ||      \      ||       \     ||      \      ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =============================================================================*/
00022 #include "kerneltypes.h"
00023 #include "ll.h"
00024 #include "threadlist.h"
00025 #include "thread.h"
00026 #include "kernel_debug.h"
00027 //---------------------------------------------------------------------------
00028 #if defined __FILE_ID__
00029     #undef __FILE_ID__
00030 #endif
00031 #define __FILE_ID__     THREADLIST_CPP
```

```
00032
00033 //---------------------------------------------------------------------------
00034 void ThreadList::SetPriority(K_UCHAR ucPriority_)
00035 {
00036     m_ucPriority = ucPriority_;
00037 }
00038
00039 //---------------------------------------------------------------------------
00040 void ThreadList::SetFlagPointer( K_UCHAR *pucFlag_)
00041 {
00042     m_pucFlag = pucFlag_;
00043 }
00044
00045 //---------------------------------------------------------------------------
00046 void ThreadList::Add(LinkListNode *node_) {
00047     CircularLinkList::Add(node_);
00048
00049     // If the head of the list isn't empty,
00050     if (m_pstHead != NULL)
00051     {
00052         // We've specified a bitmap for this threadlist
00053         if (m_pucFlag)
00054         {
00055             // Set the flag for this priority level
00056             *m_pucFlag |= (1 << m_ucPriority);
00057         }
00058     }
00059 }
00060
00061 //---------------------------------------------------------------------------
00062 void ThreadList::Add(LinkListNode *node_, K_UCHAR *pucFlag_, K_UCHAR ucPriority_
    ) {
00063     // Set the threadlist's priority level, flag pointer, and then add the
00064     // thread to the threadlist
00065     SetPriority(ucPriority_);
00066     SetFlagPointer(pucFlag_);
00067     Add(node_);
00068 }
00069
00070 //---------------------------------------------------------------------------
00071 void ThreadList::Remove(LinkListNode *node_) {
00072     // Remove the thread from the list
00073     CircularLinkList::Remove(node_);
00074
00075     // If the list is empty...
00076     if (!m_pstHead)
00077     {
00078         // Clear the bit in the bitmap at this priority level
00079         if (m_pucFlag)
00080         {
00081             *m_pucFlag &= ~(1 << m_ucPriority);
00082         }
00083     }
00084 }
00085
00086 //---------------------------------------------------------------------------
00087 Thread *ThreadList::HighestWaiter()
00088 {
00089     Thread *pclTemp = static_cast<Thread*>(GetHead());
00090     Thread *pclChosen = pclTemp;
00091
00092     K_UCHAR ucMaxPri = 0;
00093
00094     // Go through the list, return the highest-priority thread in this list.
00095     while(1)
00096     {
00097         // Compare against current max-priority thread
00098         if (pclTemp->GetPriority() >= ucMaxPri)
00099         {
00100             ucMaxPri = pclTemp->GetPriority();
00101             pclChosen = pclTemp;
00102         }
00103
00104         // Break out if this is the last thread in the list
00105         if (pclTemp == static_cast<Thread*>(GetTail()))
00106         {
00107             break;
00108         }
00109
00110         pclTemp = static_cast<Thread*>(pclTemp->GetNext());
00111     }
00112     return pclChosen;
00113 }
```

15.79 threadlist.h File Reference

Thread linked-list declarations.

```
#include "kerneltypes.h"
#include "ll.h"
```

Classes

- class ThreadList

  This class is used for building thread-management facilities, such as schedulers, and blocking objects.

15.79.1 Detailed Description

Thread linked-list declarations.

Definition in file threadlist.h.

15.80 threadlist.h

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__   |__   ___| __  |__   _______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \     ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  ___||__|\__\  ___||__|\__\  ___||_______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #ifndef __THREADLIST_H__
00023 #define __THREADLIST_H__
00024
00025 #include "kerneltypes.h"
00026 #include "ll.h"
00027
00028 class Thread;
00029
00034 class ThreadList : public CircularLinkList
00035 {
00036 public:
00040     ThreadList() { m_ucPriority = 0; m_pucFlag = NULL; }
00041
00049     void SetPriority(K_UCHAR ucPriority_);
00050
00059     void SetFlagPointer(K_UCHAR *pucFlag_);
00060
00068     void Add(LinkListNode *node_);
00069
00083     void Add(LinkListNode *node_, K_UCHAR *pucFlag_, K_UCHAR ucPriority_);
00084
00092     void Remove(LinkListNode *node_);
00093
00101     Thread *HighestWaiter();
00102 private:
00103
00105     K_UCHAR m_ucPriority;
00106
00108     K_UCHAR *m_pucFlag;
00109 };
00110
00111 #endif
00112
```

15.81 threadport.cpp File Reference

ATMega328p Multithreading.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "threadport.h"
#include "kernelswi.h"
#include "kerneltimer.h"
#include "timerlist.h"
#include "quantum.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

Functions

- static void **Thread_Switch** (void)
- ISR (INT0_vect) __attribute__((signal

  SWI using INT0 - used to trigger a context switch.
- ISR (TIMER1_COMPA_vect)

  Timer interrupt ISR - causes a tick, which may cause a context switch.

Variables

- Thread * **g_pstCurrentThread**
- **naked**

15.81.1 Detailed Description

ATMega328p Multithreading.

Definition in file threadport.cpp.

15.82 threadport.cpp

```
00001 /*===========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__   |__   ___| __  |__  ______
00004 |    \  /  | ||    \       ||      |      ||  |/ /     ||___   |
00005 |     \/   | ||     \      ||      \      ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024 #include "thread.h"
00025 #include "threadport.h"
00026 #include "kernelswi.h"
00027 #include "kerneltimer.h"
00028 #include "timerlist.h"
00029 #include "quantum.h"
00030 #include <avr/io.h>
00031 #include <avr/interrupt.h>
00032
00033 //---------------------------------------------------------------------------
00034 Thread *g_pstCurrentThread;
```

```
00035
00036 //---------------------------------------------------------------------------
00037 void ThreadPort::InitStack(Thread *pclThread_)
00038 {
00039     // Initialize the stack for a Thread
00040     K_USHORT usAddr;
00041     K_UCHAR *pucStack;
00042     K_USHORT i;
00043
00044     // Get the address of the thread's entry function
00045     usAddr = (K_USHORT)(pclThread_->m_pfEntryPoint);
00046
00047     // Start by finding the bottom of the stack
00048     pucStack = (K_UCHAR*)pclThread_->m_pwStackTop;
00049
00050     // clear the stack, and initialize it to a known-default value (easier
00051     // to debug when things go sour with stack corruption or overflow)
00052     for (i = 0; i < pclThread_->m_usStackSize; i++)
00053     {
00054         pclThread_->m_pwStack[i] = 0xFF;
00055     }
00056
00057     // Our context starts with the entry function
00058     PUSH_TO_STACK(pucStack, (K_UCHAR)(usAddr & 0x00FF));
00059     PUSH_TO_STACK(pucStack, (K_UCHAR)((usAddr >> 8) & 0x00FF));
00060
00061     // R0
00062     PUSH_TO_STACK(pucStack, 0x00);    // R0
00063
00064     // Push status register and R1 (which is used as a constant zero)
00065     PUSH_TO_STACK(pucStack, 0x80);  // SR
00066     PUSH_TO_STACK(pucStack, 0x00);  // R1
00067
00068     // Push other registers
00069     for (i = 2; i <= 23; i++) //R2-R23
00070     {
00071         PUSH_TO_STACK(pucStack, i);
00072     }
00073
00074     // Assume that the argument is the only stack variable
00075     PUSH_TO_STACK(pucStack, (K_UCHAR)(((K_USHORT)(pclThread_->
      m_pvArg)) & 0x00FF));    //R24
00076     PUSH_TO_STACK(pucStack, (K_UCHAR)((((K_USHORT)(pclThread_->
      m_pvArg))>>8) & 0x00FF)); //R25
00077
00078     // Push the rest of the registers in the context
00079     for (i = 26; i <=31; i++)
00080     {
00081         PUSH_TO_STACK(pucStack, i);
00082     }
00083
00084     // Set the top o' the stack.
00085     pclThread_->m_pwStackTop = (K_UCHAR*)pucStack;
00086
00087     // That's it!  the thread is ready to run now.
00088 }
00089
00090 //---------------------------------------------------------------------------
00091 static void Thread_Switch(void)
00092 {
00093     g_pstCurrent = g_pstNext;
00094 }
00095
00096
00097 //---------------------------------------------------------------------------
00098 void ThreadPort::StartThreads()
00099 {
00100     KernelSWI::Config();                // configure the task switch SWI
00101     KernelTimer::Config();                // configure the kernel timer
00102
00103     Scheduler::SetScheduler(1);              // enable the scheduler
00104     Scheduler::Schedule();                // run the scheduler - determine the first
       thread to run
00105
00106     Thread_Switch();                    // Set the next scheduled thread to the current thread
00107
00108     KernelTimer::Start();                // enable the kernel timer
00109     KernelSWI::Start();                    // enable the task switch SWI
00110
00111     // Restore the context...
00112     Thread_RestoreContext();        // restore the context of the first running thread
00113     ASM("reti");                    // return from interrupt - will return to the first scheduled thread
00114 }
00115
00116 //---------------------------------------------------------------------------
00121 //---------------------------------------------------------------------------
00122 ISR(INT0_vect) __attribute__ ( ( signal, naked ) );
```

```
00123 ISR(INT0_vect)
00124 {
00125     Thread_SaveContext();        // Push the context (registers) of the current task
00126     Thread_Switch();            // Switch to the next task
00127     Thread_RestoreContext();    // Pop the context (registers) of the next task
00128     ASM("reti");                // Return to the next task
00129 }
00130
00131 //------------------------------------------------------------------------
00136 //------------------------------------------------------------------------
00137 ISR(TIMER1_COMPA_vect)
00138 {
00139 #if KERNEL_USE_TIMERS
00140     TimerScheduler::Process();
00141 #endif
00142 #if KERNEL_USE_QUANTUM
00143     Quantum::UpdateTimer();
00144 #endif
00145 }
```

15.83 threadport.h File Reference

ATMega328p Multithreading support.

```
#include "kerneltypes.h"
#include "thread.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

Classes

- class ThreadPort

  Class defining the architecture specific functions required by the kernel.

Macros

- #define ASM(x) asm volatile(x);

  ASM Macro - simplify the use of ASM directive in C.
- #define SR_ 0x3F

  Status register define - map to 0x003F.
- #define SPH_ 0x3E

  Stack pointer define.
- #define **SPL_** 0x3D
- #define TOP_OF_STACK(x, y) (K_UCHAR*) (((K_USHORT)x) + (y-1))

  Macro to find the top of a stack given its size and top address.
- #define PUSH_TO_STACK(x, y) *x = y; x--;

  Push a value y to the stack pointer x and decrement the stack pointer.
- #define Thread_SaveContext()

  Save the context of the Thread.
- #define Thread_RestoreContext()

  Restore the context of the Thread.
- #define CS_ENTER()

  These macros must be used in pairs !
- #define CS_EXIT()

  Exit critical section (restore status register)
- #define ENABLE_INTS() ASM("sei");

  Initiate a contex switch without using the SWI.
- #define **DISABLE_INTS**() ASM("cli");

15.83.1 Detailed Description

ATMega328p Multithreading support.

Definition in file threadport.h.

15.83.2 Macro Definition Documentation

15.83.2.1 #define CS_ENTER( )

Value:

```
{ \
volatile K_UCHAR x; \
x = _SFR_IO8(SR_); \
ASM("cli");
```

These macros *must* be used in pairs !

Enter critical section (copy status register, disable interrupts)

Definition at line 142 of file threadport.h.

15.83.2.2 #define CS_EXIT( )

Value:

```
_SFR_IO8(SR_) = x;\
}
```

Exit critical section (restore status register)

Definition at line 149 of file threadport.h.

15.84 threadport.h

```
00001 /*=========================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   ___|_    |__   ___|__   |__   ___| __   |__   ______
00004 |    \  /   | ||     \       ||      |      ||   |/ /      ||___    |
00005 |     \/    | ||      \      ||      \      ||      \      ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========================================================================*/
00021 #ifndef __THREADPORT_H_
00022 #define __THREADPORT_H_
00023
00024 #include "kerneltypes.h"
00025 #include "thread.h"
00026
00027 #include <avr/io.h>
00028 #include <avr/interrupt.h>
00029
00030 //---------------------------------------------------------------------------
00032 #define ASM(x)      asm volatile(x);
00033 #define SR_         0x3F
00035 #define SPH_        0x3E
00037 #define SPL_        0x3D
00038
00039
00040 //---------------------------------------------------------------------------
00042 #define TOP_OF_STACK(x, y)        (K_UCHAR*) ( ((K_USHORT)x) + (y-1) )
00043 #define PUSH_TO_STACK(x, y)        *x = y; x--;
00045
```

```
00046 //---------------------------------------------------------------------------
00048 #define Thread_SaveContext() \
00049 ASM("push r0"); \
00050 ASM("in r0, __SREG__"); \
00051 ASM("cli"); \
00052 ASM("push r0"); \
00053 ASM("push r1"); \
00054 ASM("clr r1"); \
00055 ASM("push r2"); \
00056 ASM("push r3"); \
00057 ASM("push r4"); \
00058 ASM("push r5"); \
00059 ASM("push r6"); \
00060 ASM("push r7"); \
00061 ASM("push r8"); \
00062 ASM("push r9"); \
00063 ASM("push r10"); \
00064 ASM("push r11"); \
00065 ASM("push r12"); \
00066 ASM("push r13"); \
00067 ASM("push r14"); \
00068 ASM("push r15"); \
00069 ASM("push r16"); \
00070 ASM("push r17"); \
00071 ASM("push r18"); \
00072 ASM("push r19"); \
00073 ASM("push r20"); \
00074 ASM("push r21"); \
00075 ASM("push r22"); \
00076 ASM("push r23"); \
00077 ASM("push r24"); \
00078 ASM("push r25"); \
00079 ASM("push r26"); \
00080 ASM("push r27"); \
00081 ASM("push r28"); \
00082 ASM("push r29"); \
00083 ASM("push r30"); \
00084 ASM("push r31"); \
00085 ASM("lds r26, g_pstCurrent"); \
00086 ASM("lds r27, g_pstCurrent + 1"); \
00087 ASM("adiw r26, 4"); \
00088 ASM("in    r0, 0x3D"); \
00089 ASM("st    x+, r0"); \
00090 ASM("in    r0, 0x3E"); \
00091 ASM("st    x+, r0");
00092
00093 //---------------------------------------------------------------------------
00095 #define Thread_RestoreContext() \
00096 ASM("lds r26, g_pstCurrent"); \
00097 ASM("lds r27, g_pstCurrent + 1");\
00098 ASM("adiw r26, 4"); \
00099 ASM("ld     r28, x+"); \
00100 ASM("out 0x3D, r28"); \
00101 ASM("ld     r29, x+"); \
00102 ASM("out 0x3E, r29"); \
00103 ASM("pop r31"); \
00104 ASM("pop r30"); \
00105 ASM("pop r29"); \
00106 ASM("pop r28"); \
00107 ASM("pop r27"); \
00108 ASM("pop r26"); \
00109 ASM("pop r25"); \
00110 ASM("pop r24"); \
00111 ASM("pop r23"); \
00112 ASM("pop r22"); \
00113 ASM("pop r21"); \
00114 ASM("pop r20"); \
00115 ASM("pop r19"); \
00116 ASM("pop r18"); \
00117 ASM("pop r17"); \
00118 ASM("pop r16"); \
00119 ASM("pop r15"); \
00120 ASM("pop r14"); \
00121 ASM("pop r13"); \
00122 ASM("pop r12"); \
00123 ASM("pop r11"); \
00124 ASM("pop r10"); \
00125 ASM("pop r9"); \
00126 ASM("pop r8"); \
00127 ASM("pop r7"); \
00128 ASM("pop r6"); \
00129 ASM("pop r5"); \
00130 ASM("pop r4"); \
00131 ASM("pop r3"); \
00132 ASM("pop r2"); \
00133 ASM("pop r1"); \
00134 ASM("pop r0"); \
```

```
00135 ASM("out __SREG__, r0"); \
00136 ASM("pop r0");
00137
00138 //---------------------------------------------------------------------------
00140 //---------------------------------------------------------------------------
00142 #define CS_ENTER()    \
00143 { \
00144 volatile K_UCHAR x; \
00145 x = _SFR_IO8(SR_); \
00146 ASM("cli");
00147 //---------------------------------------------------------------------------
00149 #define CS_EXIT() \
00150 _SFR_IO8(SR_) = x;\
00151 }
00152
00153 //---------------------------------------------------------------------------
00155 #define ENABLE_INTS()        ASM("sei");
00156 #define DISABLE_INTS()       ASM("cli");
00157
00158 //---------------------------------------------------------------------------
00159 class Thread;
00167 class ThreadPort
00168 {
00169 public:
00175     static void StartThreads();
00176     friend class Thread;
00177 private:
00178
00186     static void InitStack(Thread *pstThread_);
00187 };
00188
00189 #endif //__ThreadPORT_H_
```

15.85 timerlist.cpp File Reference

Timer data structure + scheduler implementations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "timerlist.h"
#include "kerneltimer.h"
#include "threadport.h"
#include "kernel_debug.h"
#include "quantum.h"
```

Macros

- #define **__FILE_ID__** TIMERLIST_CPP

15.85.1 Detailed Description

Timer data structure + scheduler implementations.

Definition in file timerlist.cpp.

15.86 timerlist.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|     |_   __|_    |__  __|__   |__  __| __  |__  ______
00004 |    \   /  | ||    \      ||      |     ||  |/ /     ||___   |
00005 |     \/    | ||     \     ||      \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]-----------------------------------------------
00010
```

```
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "timerlist.h"
00026 #include "kerneltimer.h"
00027 #include "threadport.h"
00028 #include "kernel_debug.h"
00029 #include "quantum.h"
00030 //---------------------------------------------------------------------------
00031 #if defined __FILE_ID__
00032     #undef __FILE_ID__
00033 #endif
00034 #define __FILE_ID__     TIMERLIST_CPP
00035
00036 #if KERNEL_USE_TIMERS
00037
00038 //---------------------------------------------------------------------------
00039 TimerList TimerScheduler::m_clTimerList;
00040
00041 //---------------------------------------------------------------------------
00042 void TimerList::Init(void)
00043 {
00044     m_bTimerActive = 0;
00045     m_ulNextWakeup = 0;
00046 }
00047
00048 //---------------------------------------------------------------------------
00049 void TimerList::Add(Timer *pclListNode_)
00050 {
00051 #if KERNEL_TIMERS_TICKLESS
00052     bool bStart = 0;
00053 #endif
00054
00055     K_LONG lDelta;
00056     CS_ENTER();
00057
00058 #if KERNEL_TIMERS_TICKLESS
00059     if (GetHead() == NULL)
00060     {
00061         bStart = 1;
00062     }
00063 #endif
00064
00065     pclListNode_->ClearNode();
00066     DoubleLinkList::Add(pclListNode_);
00067
00068     // Set the initial timer value
00069     pclListNode_->m_ulTimeLeft = pclListNode_->m_ulInterval;
00070
00071 #if KERNEL_TIMERS_TICKLESS
00072     if (!bStart)
00073     {
00074         // If the new interval is less than the amount of time remaining...
00075         lDelta = KernelTimer::TimeToExpiry() - pclListNode_->
      m_ulInterval;
00076
00077         if (lDelta > 0)
00078         {
00079             // Set the new expiry time on the timer.
00080             m_ulNextWakeup = KernelTimer::SubtractExpiry((K_ULONG)
      lDelta);
00081         }
00082     }
00083     else
00084     {
00085         m_ulNextWakeup = pclListNode_->m_ulInterval;
00086         KernelTimer::SetExpiry(m_ulNextWakeup);
00087         KernelTimer::Start();
00088     }
00089 #endif
00090
00091     // Set the timer as active.
00092     pclListNode_->m_ucFlags |= TIMERLIST_FLAG_ACTIVE;
00093     CS_EXIT();
00094 }
00095
00096 //---------------------------------------------------------------------------
00097 void TimerList::Remove(Timer *pclLinkListNode_)
00098 {
00099     CS_ENTER();
00100
00101     DoubleLinkList::Remove(pclLinkListNode_);
00102
00103 #if KERNEL_TIMERS_TICKLESS
```

```
00104     if (this->GetHead() == NULL)
00105     {
00106         KernelTimer::Stop();
00107     }
00108 #endif
00109 
00110     CS_EXIT();
00111 }
00112 
00113 //---------------------------------------------------------------------------
00114 void TimerList::Process(void)
00115 {
00116 #if KERNEL_TIMERS_TICKLESS
00117     K_ULONG ulNewExpiry;
00118     K_ULONG ulOvertime;
00119     bool bContinue;
00120 #endif
00121 
00122     Timer *pclNode;
00123     Timer *pclPrev;
00124 
00125 #if KERNEL_USE_QUANTUM
00126     Quantum::SetInTimer();
00127 #endif
00128 #if KERNEL_TIMERS_TICKLESS
00129     // Clear the timer and its expiry time - keep it running though
00130     KernelTimer::ClearExpiry();
00131     do
00132     {
00133 #endif
00134         pclNode = static_cast<Timer*>(GetHead());
00135         pclPrev = NULL;
00136 
00137 #if KERNEL_TIMERS_TICKLESS
00138         bContinue = 0;
00139         ulNewExpiry = MAX_TIMER_TICKS;
00140 #endif
00141 
00142         // Subtract the elapsed time interval from each active timer.
00143         while (pclNode)
00144         {
00145             // Active timers only...
00146             if (pclNode->m_ucFlags & TIMERLIST_FLAG_ACTIVE)
00147             {
00148                 // Did the timer expire?
00149 #if KERNEL_TIMERS_TICKLESS
00150                 if (pclNode->m_ulTimeLeft <= m_ulNextWakeup)
00151 #else
00152                 pclNode->m_ulTimeLeft--;
00153                 if (0 == pclNode->m_ulTimeLeft)
00154 #endif
00155                 {
00156                     // Yes - set the "callback" flag - we'll execute the callbacks later
00157                     pclNode->m_ucFlags |= TIMERLIST_FLAG_CALLBACK;
00158 
00159                     if (pclNode->m_ucFlags & TIMERLIST_FLAG_ONE_SHOT)
00160                     {
00161                         // If this was a one-shot timer, deactivate the timer.
00162                         pclNode->m_ucFlags |= TIMERLIST_FLAG_EXPIRED;
00163                         pclNode->m_ucFlags &= ~TIMERLIST_FLAG_ACTIVE;

00164                     }
00165                     else
00166                     {
00167                         // Reset the interval timer.
00169                         // I think we're good though...
00170                         pclNode->m_ulTimeLeft = pclNode->
      m_ulInterval;
00171 
00172 #if KERNEL_TIMERS_TICKLESS
00173                         // If the time remaining (plus the length of the tolerance interval)
00174                         // is less than the next expiry interval, set the next expiry interval.
00175                         if ((pclNode->m_ulTimeLeft + pclNode->
      m_ulTimerTolerance) < ulNewExpiry)
00176                         {
00177                             ulNewExpiry = pclNode->m_ulTimeLeft + pclNode->
      m_ulTimerTolerance;
00178                         }
00179 #endif
00180                     }
00181                 }
00182 #if KERNEL_TIMERS_TICKLESS
00183                 else
00184                 {
00185                     // Not expiring, but determine how K_LONG to run the next timer interval for.
00186                     pclNode->m_ulTimeLeft -= m_ulNextWakeup;
00187                     if (pclNode->m_ulTimeLeft < ulNewExpiry)
```

```
00188                     {
00189                         ulNewExpiry = pclNode->m_ulTimeLeft;
00190                     }
00191                 }
00192 #endif
00193             }
00194             pclNode = static_cast<Timer*>(pclNode->GetNext());
00195         }
00196 
00197         // Process the expired timers callbacks.
00198         pclNode = static_cast<Timer*>(GetHead());
00199         while (pclNode)
00200         {
00201             pclPrev = NULL;
00202 
00203             // If the timer expired, run the callbacks now.
00204             if (pclNode->m_ucFlags & TIMERLIST_FLAG_CALLBACK)
00205             {
00206                 // Run the callback. these callbacks must be very fast...
00207                 pclNode->m_pfCallback( pclNode->m_pclOwner, pclNode->
      m_pvData );
00208                 pclNode->m_ucFlags &= ~TIMERLIST_FLAG_CALLBACK;
00209 
00210                 // If this was a one-shot timer, let's remove it.
00211                 if (pclNode->m_ucFlags & TIMERLIST_FLAG_ONE_SHOT)
00212                 {
00213                     pclPrev = pclNode;
00214                 }
00215             }
00216             pclNode = static_cast<Timer*>(pclNode->GetNext());
00217 
00218             // Remove one-shot-timers
00219             if (pclPrev)
00220             {
00221                 Remove(pclPrev);
00222             }
00223         }
00224 
00225 #if KERNEL_TIMERS_TICKLESS
00226         // Check to see how much time has elapsed since the time we
00227         // acknowledged the interrupt...
00228         ulOvertime = KernelTimer::GetOvertime();
00229 
00230         if( ulOvertime >= ulNewExpiry ) {
00231             m_ulNextWakeup = ulOvertime;
00232             bContinue = 1;
00233         }
00234 
00235     // If it's taken longer to go through this loop than would take us to
00236     // the next expiry, re-run the timing loop
00237 
00238     } while (bContinue);
00239 
00240     // This timer elapsed, but there's nothing more to do...
00241     // Turn the timer off.
00242     if (ulNewExpiry >= MAX_TIMER_TICKS)
00243     {
00244         KernelTimer::Stop();
00245     }
00246     else
00247     {
00248         // Update the timer with the new "Next Wakeup" value, plus whatever
00249         // overtime has accumulated since the last time we called this handler
00250         m_ulNextWakeup = KernelTimer::SetExpiry(ulNewExpiry +
      ulOvertime);
00251     }
00252 #endif
00253 #if KERNEL_USE_QUANTUM
00254     Quantum::ClearInTimer();
00255 #endif
00256 }
00257 
00258 //---------------------------------------------------------------------------
00259 void Timer::Start( bool bRepeat_, K_ULONG ulIntervalMs_, TimerCallback_t pfCallback_, void *
      pvData_ )
00260 {
00261     SetIntervalMSeconds(ulIntervalMs_);
00262     m_pfCallback = pfCallback_;
00263     m_pvData = pvData_;
00264     if (!bRepeat_)
00265     {
00266         m_ucFlags = TIMERLIST_FLAG_ONE_SHOT;
00267     }
00268     else
00269     {
00270         m_ucFlags = 0;
00271     }
```

```
00272     m_pclOwner = Scheduler::GetCurrentThread();
00273     TimerScheduler::Add(this);
00274 }
00275
00276 //---------------------------------------------------------------------------
00277 void Timer::Start( bool bRepeat_, K_ULONG ulIntervalMs_, K_ULONG ulToleranceMs_,
      TimerCallback_t pfCallback_, void *pvData_ )
00278 {
00279     m_ulTimerTolerance = MSECONDS_TO_TICKS(ulToleranceMs_);
00280     Start(bRepeat_, ulIntervalMs_, pfCallback_, pvData_);
00281 }
00282
00283 //---------------------------------------------------------------------------
00284 void Timer::Stop()
00285 {
00286     TimerScheduler::Remove(this);
00287 }
00288
00289 //---------------------------------------------------------------------------
00290 void Timer::SetIntervalTicks( K_ULONG ulTicks_ )
00291 {
00292     m_ulInterval = ulTicks_;
00293 }
00294
00295 //---------------------------------------------------------------------------
00297 //---------------------------------------------------------------------------
00298 void Timer::SetIntervalSeconds( K_ULONG ulSeconds_)
00299 {
00300     m_ulInterval = SECONDS_TO_TICKS(ulSeconds_);
00301 }
00302
00303 //---------------------------------------------------------------------------
00304 void Timer::SetIntervalMSeconds( K_ULONG ulMSeconds_)
00305 {
00306     m_ulInterval = MSECONDS_TO_TICKS(ulMSeconds_);
00307 }
00308
00309 //---------------------------------------------------------------------------
00310 void Timer::SetIntervalUSeconds( K_ULONG ulUSeconds_)
00311 {
00312     m_ulInterval = USECONDS_TO_TICKS(ulUSeconds_);
00313 }
00314
00315 //---------------------------------------------------------------------------
00316 void Timer::SetTolerance(K_ULONG ulTicks_)
00317 {
00318     m_ulTimerTolerance = ulTicks_;
00319 }
00320
00321
00322 #endif //KERNEL_USE_TIMERS
```

15.87 timerlist.h File Reference

Timer list and timer-scheduling declarations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
```

Classes

- class Timer

  Timer - an event-driven execution context based on a specified time interval.

- class TimerList

  TimerList class - a doubly-linked-list of timer objects.

- class TimerScheduler

  "Static" Class used to interface a global TimerList with the rest of the kernel.

Macros

- #define TIMERLIST_FLAG_ONE_SHOT (0x01)

  Timer is one-shot.
- #define TIMERLIST_FLAG_ACTIVE (0x02)

  Timer is currently active.
- #define TIMERLIST_FLAG_CALLBACK (0x04)

  Timer is pending a callback.
- #define TIMERLIST_FLAG_EXPIRED (0x08)

  Timer is actually expired.
- #define MAX_TIMER_TICKS (0x7FFFFFFF)

  Maximum value to set.
- #define **SECONDS_TO_TICKS**(x) (((K_ULONG)(x) * 1000) + 1)
- #define **MSECONDS_TO_TICKS**(x) ((K_ULONG)(x + 1))
- #define **USECONDS_TO_TICKS**(x) (((K_ULONG)(x + 999)) / 1000)
- #define MIN_TICKS (1)

  The minimum tick value to set.

Typedefs

- typedef void(* **TimerCallback_t**)(Thread *pclOwner_, void *pvData_)

15.87.1 Detailed Description

Timer list and timer-scheduling declarations. These classes implements a linked list of timer objects attached to the global kernel timer. Unlike other kernels which use a fully-synchronous "tick-based" timing mechanism, where the OS timing facilities are based on a fixed-frequency timer (which causes regular timer interrupts), Mark3 uses a "tickless" timer implementation, which only triggers interrupts when absolutely required. This is much more efficient in most cases - timer interrupts occur less frequently, allowing the kernel to stay in sleep much longer than it would otherwise.

Definition in file timerlist.h.

15.87.2 Macro Definition Documentation

15.87.2.1 #define TIMERLIST_FLAG_EXPIRED (0x08)

Timer is actually expired.

Definition at line 45 of file timerlist.h.

15.88 timerlist.h

```
00001 /*===========================================================================
00002       _____          _____          _____          _____
00003  ____|    _|__   ___|_    |__   ___|__    |__   ___| __   |__   ______
00004 |    \   /   | ||     \       ||      |     ||   |/ /      ||___    |
00005 |     \/    | ||      \      ||      \     ||      \      ||___    |
00006 |__/\__/|__|_||__|\__\   ___||__|\__\   ___||__|\__\   ___||______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00030 #ifndef __TIMERLIST_H__
00031 #define __TIMERLIST_H__
```

```
00032
00033 #include "kerneltypes.h"
00034 #include "mark3cfg.h"
00035
00036 #include "ll.h"
00037
00038 #if KERNEL_USE_TIMERS
00039 class Thread;
00040
00041 //---------------------------------------------------------------------------
00042 #define TIMERLIST_FLAG_ONE_SHOT         (0x01)
00043 #define TIMERLIST_FLAG_ACTIVE           (0x02)
00044 #define TIMERLIST_FLAG_CALLBACK         (0x04)
00045 #define TIMERLIST_FLAG_EXPIRED          (0x08)
00046
00047 //---------------------------------------------------------------------------
00048 #if KERNEL_TIMERS_TICKLESS
00049
00050 //---------------------------------------------------------------------------
00051 #define MAX_TIMER_TICKS                 (0x7FFFFFFF)
00052
00053 //---------------------------------------------------------------------------
00054 /*
00055     Ugly macros to support a wide resolution of delays.
00056     Given a 16-bit timer @ 16MHz & 256 cycle prescaler, this gives us...
00057     Max time, SECONDS_TO_TICKS:  68719s
00058     Max time, MSECONDS_TO_TICKS: 6871.9s
00059     Max time, USECONDS_TO_TICKS: 6.8719s
00060      With a 16us tick resolution.
00061 */
00062 //---------------------------------------------------------------------------
00063 #define SECONDS_TO_TICKS(x)             ((((K_ULONG)x) * TIMER_FREQ))
00064 #define MSECONDS_TO_TICKS(x)            ((((((K_ULONG)x) * (TIMER_FREQ/100)) + 5) / 10))
00065 #define USECONDS_TO_TICKS(x)            ((((((K_ULONG)x) * TIMER_FREQ) + 50000) / 1000000))
00066
00067 //---------------------------------------------------------------------------
00068 #define MIN_TICKS                        (3)
00069 //---------------------------------------------------------------------------
00070
00071 #else
00072 //---------------------------------------------------------------------------
00073 // Tick-based timers, assuming 1khz tick rate
00074 #define MAX_TIMER_TICKS                 (0x7FFFFFFF)
00075
00076 //---------------------------------------------------------------------------
00077 // add time because we don't know how far in an epoch we are when a call is made.
00078 #define SECONDS_TO_TICKS(x)             (((K_ULONG)(x) * 1000) + 1)
00079 #define MSECONDS_TO_TICKS(x)            ((K_ULONG)(x + 1))
00080 #define USECONDS_TO_TICKS(x)            (((K_ULONG)(x + 999)) / 1000)
00081
00082 //---------------------------------------------------------------------------
00083 #define MIN_TICKS                       (1)
00084 //---------------------------------------------------------------------------
00085
00086 #endif // KERNEL_TIMERS_TICKLESS
00087
00088 typedef void (*TimerCallback_t)(Thread *pclOwner_, void *pvData_);
00089
00090 //---------------------------------------------------------------------------
00091 class TimerList;
00092 class TimerScheduler;
00093 class Quantum;
00099 class Timer : public LinkListNode
00100 {
00101 public:
00105     Timer() { }
00106
00110     void Init() { ClearNode(); m_ulInterval = 0;
      m_ulTimerTolerance = 0; m_ulTimeLeft = 0;
      m_ucFlags = 0; }
00111
00121     void Start( bool bRepeat_, K_ULONG ulIntervalMs_, TimerCallback_t pfCallback_, void *pvData_ );
00122
00134     void Start( bool bRepeat_, K_ULONG ulIntervalMs_, K_ULONG ulToleranceMs_, TimerCallback_t
      pfCallback_, void *pvData_ );
00135
00140     void Stop();
00141
00151     void SetFlags (K_UCHAR ucFlags_) { m_ucFlags = ucFlags_; }
00152
00160     void SetCallback( TimerCallback_t pfCallback_){ m_pfCallback = pfCallback_; }
00161
00169     void SetData( void *pvData_ ){ m_pvData = pvData_; }
00170
00179     void SetOwner( Thread *pclOwner_){ m_pclOwner = pclOwner_; }
00180
00188     void SetIntervalTicks(K_ULONG ulTicks_);
```

```
00189
00197     void SetIntervalSeconds(K_ULONG ulSeconds_);
00198
00199
00200     K_ULONG GetInterval()    { return m_ulInterval; }
00201
00209     void SetIntervalMSeconds(K_ULONG ulMSeconds_);
00210
00218     void SetIntervalUSeconds(K_ULONG ulUSeconds_);
00219
00229     void SetTolerance(K_ULONG ulTicks_);
00230
00231 private:
00232
00233     friend class TimerList;
00234
00236     K_UCHAR m_ucFlags;
00237
00239     TimerCallback_t m_pfCallback;
00240
00242     K_ULONG m_ulInterval;
00243
00245     K_ULONG m_ulTimeLeft;
00246
00248     K_ULONG m_ulTimerTolerance;
00249
00251     Thread  *m_pclOwner;
00252
00254     void    *m_pvData;
00255 };
00256
00257 //---------------------------------------------------------------------------
00261 class TimerList : public DoubleLinkList
00262 {
00263 public:
00270     void Init();
00271
00279     void Add(Timer *pclListNode_);
00280
00288     void Remove(Timer *pclListNode_);
00289
00296     void Process();
00297
00298 private:
00300     K_ULONG m_ulNextWakeup;
00301
00303     K_UCHAR m_bTimerActive;
00304 };
00305
00306 //---------------------------------------------------------------------------
00311 class TimerScheduler
00312 {
00313 public:
00320     static void Init() { m_clTimerList.Init(); }
00321
00330     static void Add(Timer *pclListNode_)
00331         {m_clTimerList.Add(pclListNode_); }
00332
00341     static void Remove(Timer *pclListNode_)
00342         {m_clTimerList.Remove(pclListNode_); }
00343
00352     static void Process() {m_clTimerList.Process();}
00353 private:
00354
00356     static TimerList m_clTimerList;
00357 };
00358
00359 #endif // KERNEL_USE_TIMERS
00360
00361 #endif
```

15.89 tracebuffer.cpp File Reference

Kernel trace buffer class definition.

```
#include "kerneltypes.h"
#include "tracebuffer.h"
#include "mark3cfg.h"
#include "writebuf16.h"
#include "kernel_debug.h"
```

15.89.1 Detailed Description

Kernel trace buffer class definition.

Definition in file tracebuffer.cpp.

15.90 tracebuffer.cpp

```
00001 /*=============================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
00004 |    \  /  | ||    \       ||      |     ||  |/ /     ||___   |
00005 |     \/   | ||     \      ||      \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00019 #include "kerneltypes.h"
00020 #include "tracebuffer.h"
00021 #include "mark3cfg.h"
00022 #include "writebuf16.h"
00023 #include "kernel_debug.h"
00024
00025 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00026 //---------------------------------------------------------------------------
00027 WriteBuffer16 TraceBuffer::m_clBuffer;
00028 volatile K_USHORT TraceBuffer::m_usIndex;
00029 K_USHORT TraceBuffer::m_ausBuffer[ (TRACE_BUFFER_SIZE/sizeof(K_USHORT)) ];
00030
00031 //---------------------------------------------------------------------------
00032 void TraceBuffer::Init()
00033 {
00034     m_clBuffer.SetBuffers(m_ausBuffer, TRACE_BUFFER_SIZE/sizeof(K_USHORT));
00035     m_usIndex = 0;
00036 }
00037
00038 //---------------------------------------------------------------------------
00039 K_USHORT TraceBuffer::Increment()
00040 {
00041     return m_usIndex++;
00042 }
00043
00044 //---------------------------------------------------------------------------
00045 void TraceBuffer::Write( K_USHORT *pusData_, K_USHORT usSize_ )
00046 {
00047     // Pipe the data directly to the circular buffer
00048     m_clBuffer.WriteData(pusData_, usSize_);
00049 }
00050
00051 #endif
00052
```

15.91 tracebuffer.h File Reference

Kernel trace buffer class declaration.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "writebuf16.h"
```

15.91.1 Detailed Description

Kernel trace buffer class declaration. Global kernel trace-buffer. Used to instrument the kernel with lightweight encoded print statements. If something goes wrong, the tracebuffer can be examined for debugging purposes. Also, subsets of kernel trace information can be extracted and analyzed to provide information about runtime performance, thread-scheduling, and other nifty things in real-time.

Definition in file tracebuffer.h.

15.92 tracebuffer.h

```
00001 /*=============================================================================
00002      _____          _____          _____          _____
00003  ___|    _|__   __|_    |__   __|__    |__   __| __  |__  _______
00004 |    \  /  | ||    \       ||      |      ||  |/ /     ||___    |
00005 |     \/   | ||     \      ||      \      ||     \     ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =============================================================================*/
00024 #ifndef __TRACEBUFFER_H__
00025 #define __TRACEBUFFER_H__
00026
00027 #include "kerneltypes.h"
00028 #include "mark3cfg.h"
00029 #include "writebuf16.h"
00030
00031 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00032
00033 #define TRACE_BUFFER_SIZE           (16)
00034
00038 class TraceBuffer
00039 {
00040 public:
00046     static void Init();
00047
00055     static K_USHORT Increment();
00056
00065     static void Write( K_USHORT *pusData_, K_USHORT usSize_ );
00066
00075     void SetCallback( WriteBufferCallback pfCallback_ )
00076         { m_clBuffer.SetCallback( pfCallback_ ); }
00077 private:
00078
00079     static WriteBuffer16 m_clBuffer;
00080     static volatile K_USHORT m_usIndex;
00081     static K_USHORT m_ausBuffer[ (TRACE_BUFFER_SIZE / sizeof( K_USHORT )) ];
00082 };
00083
00084 #endif //KERNEL_USE_DEBUG
00085
00086 #endif
```

15.93 writebuf16.cpp File Reference

16 bit circular buffer implementation with callbacks.

```
#include "kerneltypes.h"
#include "writebuf16.h"
#include "kernel_debug.h"
#include "threadport.h"
```

15.93.1 Detailed Description

16 bit circular buffer implementation with callbacks.

Definition in file writebuf16.cpp.

15.94 writebuf16.cpp

```
00001 /*=========================================================================
00002      ______          ______          ______          ______
00003  ___|    _|__   __|_    |__   __|__   |__   __| __   |__   _______
00004 |    \  /   | ||     \       ||      |      ||   |/ /      ||____   |
00005 |     \/    | ||      \      ||      \      ||      \      ||____   |
00006 |__/\__/|__|_||__|\__\   __||__|\__\   __||__|\__\   __||_______|
00007     |______|       |______|       |______|       |______|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00020 #include "kerneltypes.h"
00021 #include "writebuf16.h"
00022 #include "kernel_debug.h"
00023 #include "threadport.h"
00024
00025 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00026
00027 //---------------------------------------------------------------------------
00028 void WriteBuffer16::WriteData( K_USHORT *pusBuf_, K_USHORT usLen_ )
00029 {
00030     K_USHORT *apusBuf[1];
00031     K_USHORT ausLen[1];
00032
00033     apusBuf[0] = pusBuf_;
00034     ausLen[0] = usLen_;
00035
00036     WriteVector( apusBuf, ausLen, 1 );
00037 }
00038
00039 //---------------------------------------------------------------------------
00040 void WriteBuffer16::WriteVector( K_USHORT **ppusBuf_, K_USHORT *pusLen_, K_UCHAR ucCount_ )
00041 {
00042     K_USHORT usTempHead;
00043     K_UCHAR i;
00044     K_UCHAR j;
00045     K_USHORT usTotalLen = 0;
00046     bool bCallback = false;
00047     bool bRollover = false;
00048     // Update the head pointer synchronously, using a small
00049     // critical section in order to provide thread safety without
00050     // compromising on responsiveness by adding lots of extra
00051     // interrupt latency.
00052
00053     CS_ENTER();
00054
00055     usTempHead = m_usHead;
00056     {
00057         for (i = 0; i < ucCount_; i++)
00058         {
00059             usTotalLen += pusLen_[i];
00060         }
00061         m_usHead = (usTempHead + usTotalLen) % m_usSize;
00062     }
00063     CS_EXIT();
00064
00065     // Call the callback if we cross the 50% mark or rollover
00066     if (m_usHead < usTempHead)
00067     {
00068         if (m_pfCallback)
00069         {
00070             bCallback = true;
00071             bRollover = true;
00072         }
00073     }
00074     else if ((usTempHead < (m_usSize >> 1)) && (m_usHead >= (m_usSize >> 1)))
00075     {
00076         // Only trigger the callback if it's non-null
00077         if (m_pfCallback)
00078         {
00079             bCallback = true;
00080         }
00081     }
00082
00083     // Are we going to roll-over?
00084     for (j = 0; j < ucCount_; j++)
00085     {
```

```
00086         K_USHORT usSegmentLength = pusLen_[j];
00087         if (usSegmentLength + usTempHead >= m_usSize)
00088         {
00089             // We need to two-part this... First part: before the rollover
00090             K_USHORT usTempLen;
00091             K_USHORT *pusTmp = &m_pusData[ usTempHead ];
00092             K_USHORT *pusSrc = ppusBuf_[j];
00093             usTempLen = m_usSize - usTempHead;
00094             for (i = 0; i < usTempLen; i++)
00095             {
00096                 *pusTmp++ = *pusSrc++;
00097             }
00098 
00099             // Second part: after the rollover
00100             usTempLen = usSegmentLength - usTempLen;
00101             pusTmp = m_pusData;
00102             for (i = 0; i < usTempLen; i++)
00103             {
00104                 *pusTmp++ = *pusSrc++;
00105             }
00106         }
00107         else
00108         {
00109             // No rollover - do the copy all at once.
00110             K_USHORT *pusSrc = ppusBuf_[j];
00111             K_USHORT *pusTmp = &m_pusData[ usTempHead ];
00112             for (K_USHORT i = 0; i < usSegmentLength; i++)
00113             {
00114                 *pusTmp++ = *pusSrc++;
00115             }
00116         }
00117     }
00118 
00119 
00120     // Call the callback if necessary
00121     if (bCallback)
00122     {
00123         if (bRollover)
00124         {
00125             // Rollover - process the back-half of the buffer
00126             m_pfCallback( &m_pusData[ m_usSize >> 1], m_usSize >> 1 );
00127         }
00128         else
00129         {
00130             // 50% point - process the front-half of the buffer
00131             m_pfCallback( m_pusData, m_usSize >> 1);
00132         }
00133     }
00134 }
00135 
00136 #endif
```

15.95 writebuf16.h File Reference

Thread-safe circular buffer implementation with 16-bit elements.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
```

15.95.1 Detailed Description

Thread-safe circular buffer implementation with 16-bit elements.

Definition in file writebuf16.h.

15.96 writebuf16.h

```
00001 /*===========================================================================
00002       _____          _____          _____          _____
00003  ____|    _|__   ____|_    |__   ____|__    |__   ____|__  |__   ______
00004 |    \  /   | ||    \      ||     |     ||   |/ /     ||___   |
00005 |     \/    | ||     \     ||     \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
```

```
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2015 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00020 #ifndef __WRITEBUF16_H__
00021 #define __WRITEBUF16_H__
00022
00023 #include "kerneltypes.h"
00024 #include "mark3cfg.h"
00025
00026 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00027
00032 typedef void (*WriteBufferCallback)( K_USHORT *pusData_, K_USHORT usSize_ );
00033
00040 class WriteBuffer16
00041 {
00042 public:
00053     void SetBuffers( K_USHORT *pusData_, K_USHORT usSize_ )
00054     {
00055         m_pusData = pusData_;
00056         m_usSize = usSize_;
00057         m_usHead = 0;
00058         m_usTail = 0;
00059     }
00060
00072     void SetCallback( WriteBufferCallback pfCallback_ )
00073         { m_pfCallback = pfCallback_; }
00074
00083     void WriteData( K_USHORT *pusBuf_, K_USHORT usLen_ );
00084
00094     void WriteVector( K_USHORT **ppusBuf_, K_USHORT *pusLen_, K_UCHAR ucCount_);
00095
00096 private:
00097     K_USHORT *m_pusData;
00098
00099     volatile K_USHORT m_usSize;
00100     volatile K_USHORT m_usHead;
00101     volatile K_USHORT m_usTail;
00102
00103     WriteBufferCallback m_pfCallback;
00104 };
00105 #endif
00106
00107 #endif
```

Index