Mark3 Realtime Kernel

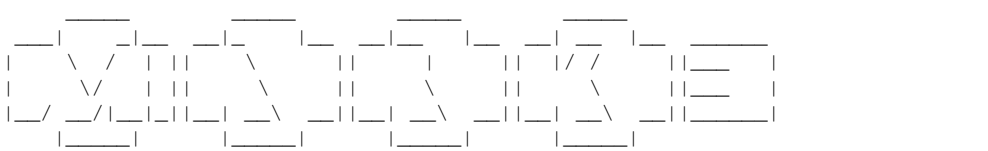
Generated by Doxygen 1.8.7

Mon Oct 5 2015 21:28:09

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 u16 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 u16 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 u8/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 R↩ AM - 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, u16B, etc. All of which can be configured to provide a stimulus to your system by means of interrupts. This stimulus gives u16 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 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 u16 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 uint8_t 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 popu32ar 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 uint8_t hp_tasks = 0;

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

volatile uint8_t 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 uint8_t aucAppStack[STACK_SIZE_APP];
static uint8_t 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 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;
uint8_t 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)
    {
        ...
        uint16_t u16WakeCondition;

        // Allow this thread to block on multiple flags
        u16WakeCondition = clEventFlag.Wait(0x00FF, EVENT_FLAG_ANY);

        // Clear the event condition that caused the thread to wake (in this case,
        // u16WakeCondtion will equal 0x20 when triggered from the interrupt above)
        clEventFlag.Clear(u16WakeCondition);

        // <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 Mailboxes

Another form of IPC is provided by Mark3, in the form of Mailboxes and Envelopes.

Mailboxes are similar to message queues in that they provide a synchronized interface by which data can be transmitted between threads.

Where Message Queues rely on linked lists of lightweight message objects (containing only message code and a void* data-pointer), which are inherently abstract, Mailboxes use a dedicated blob of memory, which is carved up into fixed-size chunks called Envelopes (defined by the user), which are sent and received. Unlike message queues, mailbox data is copied to and from the mailboxes dedicated pool.

Mailboxes also differ in that they provide not only a blocking "receive" call, but also a blocking "send" call, providing the opportunity for threads to block on "mailbox full" as well as "mailbox empty" conditions.

All send/receive APIs support an optional timeout parameter if the KERNEL_USE_TIMEOUTS option has been configured in mark3cfg.h

6.8.1 Mailbox Example

```
// Create a mailbox object, and define a buffer that will be used to store the
// mailbox' envelopes.
static Mailbox clMbox;
static uint8_t aucMBoxBuffer[128];

...
void InitMailbox(void)
{
    // Initialize our mailbox, telling it to use our defined buffer for envelope
    // storage.  Pass in the size of the buffer, and set the size of each
    // envelope to 16 bytes.  This gives u16 a mailbox capacity of (128 / 16) = 8
    // envelopes.
    clMbox.Init((void*)aucMBoxBuffer, 128, 16);

}

...
void SendThread(void)
{
    // Define a buffer that we'll eventually send to the
    // mailbox.  Note the size is the same as that of an
    // envelope.
    uint8_t aucTxBuf[16];

    while(1)
    {
        // Copy some data into aucTxBuf, a 16-byte buffer, the
        // same size as a mailbox envelope.
        ...

        // Deliver the envelope (our buffer) into the mailbox
        clMbox.Send((void*)aucTxBuf);
    }
}

...
void RecvThred(void)
{
    uint8_t aucRxBuf[16];

    while(1)
    {
        // Wait until there's a message in our mailbox.  Once
        // there is a message, read it into our local buffer.
        cmMbox.Receive((void*)aucRxBuf);
```

```
        // Do something with the contents of aucRxBuf, which now
        // contains an envelope of data read from the mailbox.
        ...
    }
}
```

6.9 Notification Objects

Notification objects are the most lightweight of all blocking objects supplied by Mark3.

using this blocking primative, one or more threads wait for the notification object to be signalled by code elsewhere in the system (i.e. another thread or interrupt). Once the the notification has been signalled, all threads currently blocked on the object become unblocked.

6.9.1 Notification Example

```
static Notify clNotifier;

...
void MyThread(void *unused_)
{
    // Initialize our notification object before use
    clNotifier.Init();

    while (1)
    {
        // Wait until our thread has been notified that it
        // can wake up.
        clNotify.Wait();

        ...
        // Thread has woken up now -- do something!
    }
}

...
void SignalCallback(void)
{
    // Something in the system (interrupt, thread event, IPC,
    // etc.,) has called this function.  As a result, we need
    // our other thread to wake up.  Call the Notify object's
    // Signal() method to wake the thread up.  Note that this
    // will have no effect if the thread is not presently
    // blocked.

    clNotify.Signal();
}
```

6.10 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;
}
```

6.11 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

Fri Sep 18 21:18:25 EDT 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 127 iterations)
- Semaphore Post (uncontested): 103 cycles (averaged over 127 iterations)
- Semaphore Pend (uncontested): 63 cycles (averaged over 127 iterations)
- Semaphore Flyback Time (Contested Pend): 1679 cycles (averaged over 127 iterations)
- Mutex Init: 215 cycles (averaged over 127 iterations)
- Mutex Claim: 247 cycles (averaged over 127 iterations)
- Mutex Release: 159 cycles (averaged over 127 iterations)
- Thread Initialize: 8367 cycles (averaged over 126 iterations)
- Thread Start: 831 cycles (averaged over 126 iterations)
- Context Switch: 175 cycles (averaged over 126 iterations)
- Thread Schedule: 71 cycles (averaged over 126 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: 241
- URL: svn+ssh://m0slevin.code.sf.net/p/mark3/source/trunk/embedded Relative URL: ^/trunk/embedded

Date Profiled: Fri Sep 18 21:18:28 EDT 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..... : 92 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)..... : 698 Bytes
- Notification Blocking Object..... : 538 Bytes
- Performance-profiling timers..... : 546 Bytes
- Round-Robin Scheduling Support..... : 264 Bytes
- Thread Scheduling..... : 452 Bytes
- Semaphore (Synchronization Object)..... : 540 Bytes
- Mailbox IPC Support..... : 966 Bytes
- Thread Implementation..... : 1611 Bytes
- Fundamental Kernel Thread-list Data Structures.. : 210 Bytes
- Mark3 Kernel Base Class..... : 110 Bytes
- Software Timer Kernel Object..... : 378 Bytes
- Software Timer Management..... : 645 Bytes
- Runtime Kernel Trace Implementation..... : 0 Bytes
- Circular Logging Buffer Base Class..... : 0 Bytes
- Atmel AVR - Kernel Aware Simulation Support..... : 296 Bytes
- Atmel AVR - Basic Threading Support..... : 598 Bytes
- Atmel AVR - Kernel Interrupt Implemenation..... : 56 Bytes
- Atmel AVR - Kernel Timer Implementation..... : 322 Bytes
- kernelprofile.cpp.o : 256 Bytes

Mark3 Kernel Size Summary:

- Kernel : 2971 Bytes
- Synchronization Objects : 2434 Bytes
- Port : 4672 Bytes
- Features : 2059 Bytes
- Total Size : 12136 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

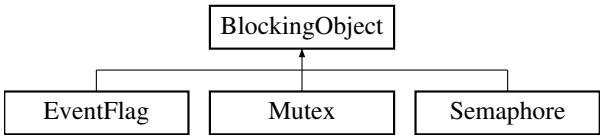
Class Documentation

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 41 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 57 of file blocking.cpp.

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

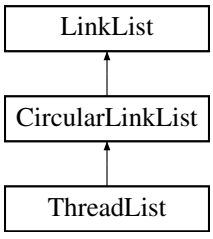
- /home/vm/mark3/trunk/embedded/kernel/public/blocking.h
- /home/vm/mark3/trunk/embedded/kernel/blocking.cpp

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 108 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 133 of file ll.cpp.

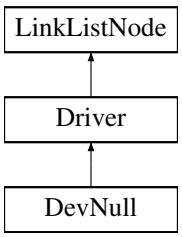
The documentation for this class was generated from the following files:

- /home/vm/mark3/trunk/embedded/kernel/public/ll.h
- /home/vm/mark3/trunk/embedded/kernel/ll.cpp

14.3 DevNull Class Reference

This class implements the "default" driver (/dev/null)

Inheritance diagram for DevNull:



Public Member Functions

- virtual void Init ()

  Initialize a driver, must be called prior to use.
- virtual uint8_t Open ()

  Open a device driver prior to use.
- virtual uint8_t Close ()

  Close a previously-opened device driver.
- virtual uint16_t Read (uint16_t u16Bytes_, uint8_t *pu8Data_)

  Read a specified number of bytes from the device into a specific buffer.
- virtual uint16_t Write (uint16_t u16Bytes_, uint8_t *pu8Data_)

  Write a payload of data of a given length to the device.
- virtual uint16_t Control (uint16_t u16Event_, void *pvDataIn_, uint16_t u16SizeIn_, void *pvDataOut_↩ , uint16_t u16SizeOut_)

  This is the main entry-point for device-specific io and control operations.

Additional Inherited Members

14.3.1 Detailed Description

This class implements the "default" driver (/dev/null)

Definition at line 46 of file driver.cpp.

14.3.2 Member Function Documentation

14.3.2.1 virtual uint8_t DevNull::Close ( ) [inline], [virtual]

Close a previously-opened device driver.

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implements Driver.

Definition at line 51 of file driver.cpp.

14.3.2.2 virtual uint16_t DevNull::Control (uint16_t *u16Event_*, void * *pvDataIn_*, uint16_t *u16SizeIn_*, void * *pvDataOut_*, uint16_t *u16SizeOut_*) [inline], [virtual]

This is the main entry-point for device-specific io and control operations.

This is used for implementing all "side-channel" communications with a device, and any device-specific IO operations that do not conform to the typical POSIX read/write paradigm. use of this funciton is analagous to the non-POSIX (yet still common) devctl() or ioctl().

Parameters

*u16Event_*	Code defining the io event (driver-specific)
*pvDataIn_*	Pointer to the intput data
*u16SizeIn_*	Size of the input data (in bytes)

*pvDataOut_*	Pointer to the output data
*u16SizeOut_*	Size of the output data (in bytes)

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implements Driver.

Definition at line 59 of file driver.cpp.

14.3.2.3 virtual uint8_t DevNull::Open ( ) [inline], [virtual]

Open a device driver prior to use.

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implements Driver.

Definition at line 50 of file driver.cpp.

14.3.2.4 virtual uint16_t DevNull::Read (uint16_t *u16Bytes_*, uint8_t * *pu8Data_*) [inline], [virtual]

Read a specified number of bytes from the device into a specific buffer.

Depending on the driver-specific implementation, this may be a number less than the requested number of bytes read, indicating that there there was less input than desired, or that as a result of buffering, the data may not be available.

Parameters

*u16Bytes_*	Number of bytes to read (<= size of the buffer)
*pu8Data_*	Pointer to a data buffer receiving the read data

Returns

Number of bytes actually read

Implements Driver.

Definition at line 53 of file driver.cpp.

14.3.2.5 virtual uint16_t DevNull::Write (uint16_t *u16Bytes_*, uint8_t * *pu8Data_*) [inline], [virtual]

Write a payload of data of a given length to the device.

Depending on the implementation of the driver, the amount of data written to the device may be less than the requested number of bytes. A result less than the requested size may indicate that the device buffer is full, indicating that the user must retry the write at a later point with the remaining data.

Parameters

*u16Bytes_*	Number of bytes to write (<= size of the buffer)

*pu8Data_*	Pointer to a data buffer containing the data to write

Returns

Number of bytes actually written

Implements Driver.

Definition at line 56 of file driver.cpp.

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

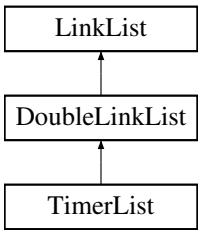
- /home/vm/mark3/trunk/embedded/kernel/driver.cpp

14.4 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

14.4.1 Detailed Description

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

Definition at line 165 of file ll.h.

14.4.2 Member Function Documentation

14.4.2.1 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 47 of file ll.cpp.

14.4.2.2 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 71 of file ll.cpp.

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

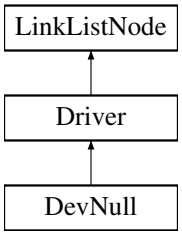
- /home/vm/mark3/trunk/embedded/kernel/public/ll.h
- /home/vm/mark3/trunk/embedded/kernel/ll.cpp

14.5 Driver Class Reference

Base device-driver class used in hardware abstraction.

`#include <driver.h>`

Inheritance diagram for Driver:



Public Member Functions

- virtual void Init ()=0

  Initialize a driver, must be called prior to use.
- virtual uint8_t Open ()=0

  Open a device driver prior to use.
- virtual uint8_t Close ()=0

  Close a previously-opened device driver.
- virtual uint16_t Read (uint16_t u16Bytes_, uint8_t *pu8Data_)=0

  Read a specified number of bytes from the device into a specific buffer.
- virtual uint16_t Write (uint16_t u16Bytes_, uint8_t *pu8Data_)=0

  Write a payload of data of a given length to the device.
- virtual uint16_t Control (uint16_t u16Event_, void *pvDataIn_, uint16_t u16SizeIn_, void *pvDataOut_↩ , uint16_t u16SizeOut_)=0

  This is the main entry-point for device-specific io and control operations.

- void SetName (const char *pcName_)

  Set the path for the driver.
- const char * GetPath ()

  Returns a string containing the device path.

Private Attributes

- const char * m_pcPath

  string pointer that holds the driver path (name)

Additional Inherited Members

14.5.1 Detailed Description

Base device-driver class used in hardware abstraction.

All other device drivers inherit from this class

Definition at line 121 of file driver.h.

14.5.2 Member Function Documentation

14.5.2.1 uint8_t Driver::Close ( ) [pure virtual]

Close a previously-opened device driver.

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implemented in DevNull.

14.5.2.2 uint16_t Driver::Control (uint16_t *u16Event_*, void * *pvDataIn_*, uint16_t *u16SizeIn_*, void * *pvDataOut_*, uint16_t *u16SizeOut_*) [pure virtual]

This is the main entry-point for device-specific io and control operations.

This is used for implementing all "side-channel" communications with a device, and any device-specific IO operations that do not conform to the typical POSIX read/write paradigm. use of this funciton is analagous to the non-POSIX (yet still common) devctl() or ioctl().

Parameters

*u16Event_*	Code defining the io event (driver-specific)
*pvDataIn_*	Pointer to the intput data
*u16SizeIn_*	Size of the input data (in bytes)
*pvDataOut_*	Pointer to the output data
*u16SizeOut_*	Size of the output data (in bytes)

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implemented in DevNull.

14.5.2.3 const char * Driver::GetPath ( ) [inline]

Returns a string containing the device path.

Returns

pcName_ Return the string constant representing the device path

Definition at line 231 of file driver.h.

14.5.2.4 uint8_t Driver::Open ( ) [pure virtual]

Open a device driver prior to use.

Returns

Driver-specific return code, 0 = OK, non-0 = error

Implemented in DevNull.

14.5.2.5 uint16_t Driver::Read (uint16_t *u16Bytes_*, uint8_t * *pu8Data_*) [pure virtual]

Read a specified number of bytes from the device into a specific buffer.

Depending on the driver-specific implementation, this may be a number less than the requested number of bytes read, indicating that there there was less input than desired, or that as a result of buffering, the data may not be available.

Parameters

*u16Bytes_*	Number of bytes to read (<= size of the buffer)
*pu8Data_*	Pointer to a data buffer receiving the read data

Returns

Number of bytes actually read

Implemented in DevNull.

14.5.2.6 void Driver::SetName (const char * *pcName_*) [inline]

Set the path for the driver.

Name must be set prior to access (since driver access is name-based).

Parameters

*pcName_*	String constant containing the device path

Definition at line 222 of file driver.h.

14.5.2.7 uint16_t Driver::Write (uint16_t *u16Bytes_*, uint8_t * *pu8Data_*) [pure virtual]

Write a payload of data of a given length to the device.

Depending on the implementation of the driver, the amount of data written to the device may be less than the requested number of bytes. A result less than the requested size may indicate that the device buffer is full, indicating that the user must retry the write at a later point with the remaining data.

Parameters

*u16Bytes_*	Number of bytes to write (<= size of the buffer)
*pu8Data_*	Pointer to a data buffer containing the data to write

Returns

Number of bytes actually written

Implemented in DevNull.

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

- /home/vm/mark3/trunk/embedded/kernel/public/driver.h

14.6 DriverList Class Reference

List of Driver objects used to keep track of all device drivers in the system.

`#include <driver.h>`

Static Public Member Functions

- static void Init ()

  Initialize the list of drivers.
- static void Add (Driver *pclDriver_)

  Add a Driver object to the managed global driver-list.
- static void Remove (Driver *pclDriver_)

  Remove a driver from the global driver list.
- static Driver * FindByPath (const char *m_pcPath)

  Look-up a driver in the global driver-list based on its path.

Static Private Attributes

- static DoubleLinkList m_clDriverList

  LinkedList object used to implementing the driver object management.

14.6.1 Detailed Description

List of Driver objects used to keep track of all device drivers in the system.

By default, the list contains a single entity, "/dev/null".

Definition at line 244 of file driver.h.

14.6.2 Member Function Documentation

14.6.2.1 DriverList::Add (Driver * *pclDriver_*) [inline], [static]

Add a Driver object to the managed global driver-list.

Parameters

*pclDriver_*	pointer to the driver object to add to the global driver list.

Definition at line 264 of file driver.h.

14.6.2.2 Driver * DriverList::FindByPath (const char * *m_pcPath*) [static]

Look-up a driver in the global driver-list based on its path.

In the event that the driver is not found in the list, a pointer to the default "/dev/null" object is returned. In this way, unimplemented drivers are automatically stubbed out.

Definition at line 113 of file driver.cpp.

14.6.2.3 void DriverList::Init ( ) [static]

Initialize the list of drivers.

Must be called prior to using the device driver library.

Definition at line 104 of file driver.cpp.

14.6.2.4 void DriverList::Remove (Driver * *pclDriver_*) [inline],[static]

Remove a driver from the global driver list.

Parameters

*pclDriver_*	Pointer to the driver object to remove from the global table

Definition at line 274 of file driver.h.

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

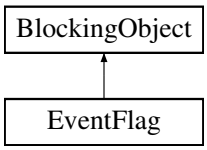
- /home/vm/mark3/trunk/embedded/kernel/public/driver.h
- /home/vm/mark3/trunk/embedded/kernel/driver.cpp

14.7 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.
- uint16_t Wait (uint16_t u16Mask_, EventFlagOperation_t eMode_)

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

- uint16_t Wait (uint16_t u16Mask_, EventFlagOperation_t eMode_, uint32_t u32TimeMS_)

  Wait - Block a thread on the specific flags in this event flag group.
- void WakeMe (Thread *pclOwner_)

  WakeMe.
- void Set (uint16_t u16Mask_)

  Set - Set additional flags in this object (logical OR).
- void Clear (uint16_t u16Mask_)

  ClearFlags - Clear a specific set of flags within this object, specific by bitmask.
- uint16_t GetMask ()

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

Private Member Functions

- uint16_t Wait_i (uint16_t u16Mask_, EventFlagOperation_t eMode_, uint32_t u32TimeMS_)

  Wait_i.

Private Attributes

- uint16_t m_u16SetMask

  Event flags currently set in this object.

Additional Inherited Members

14.7.1 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.

14.7.2 Member Function Documentation

14.7.2.1 void EventFlag::Clear (uint16_t *u16Mask_*)

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

Parameters

*u16Mask_*	- Bitmask of flags to clear

Definition at line 306 of file eventflag.cpp.

14.7.2.2 uint16_t 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 315 of file eventflag.cpp.

14.7.2.3 void EventFlag::Set (uint16_t *u16Mask_*)

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

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

Parameters

*u16Mask_*	- Bitmask of flags to set.

Definition at line 187 of file eventflag.cpp.

14.7.2.4 uint16_t EventFlag::Wait (uint16_t *u16Mask_*, EventFlagOperation_t *eMode_*)

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

Parameters

*u16Mask_*	- 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 169 of file eventflag.cpp.

14.7.2.5 uint16_t EventFlag::Wait (uint16_t *u16Mask_*, EventFlagOperation_t *eMode_*, uint32_t *u32TimeMS_*)

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

Parameters

*u16Mask_*	- 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
*u32TimeMS_*	- Time to block (in ms)

Returns

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

Definition at line 180 of file eventflag.cpp.

14.7.2.6 uint16_t EventFlag::Wait_i (uint16_t *u16Mask_*, EventFlagOperation_t *eMode_*, uint32_t *u32TimeMS_*) [private]

Wait_i.

Interal abstraction used to manage both timed and untimed wait operations

Parameters

*u16Mask_*	- 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
*u32TimeMS_*	- Time to block (in ms)

Returns

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

! 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 76 of file eventflag.cpp.

14.7.2.7 void EventFlag::WakeMe (Thread * *pclOwner_*)

WakeMe.

Wake the given thread, currently blocking on this object

Parameters

*pclOwner_*	Pointer to the owner thread to unblock.

Definition at line 68 of file eventflag.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/public/eventflag.h
- /home/vm/mark3/trunk/embedded/kernel/eventflag.cpp

14.8 FakeThread_t Struct Reference

If the kernel is set up to use an idle function instead of an idle thread, we use a placeholder data structure to "simulate" the effect of having an idle thread in the system.

`#include <thread.h>`

Public Attributes

- K_WORD * m_pwStackTop

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

  Pointer to the thread's stack.
- uint8_t m_u8ThreadID

  Thread ID.
- uint8_t m_u8Priority

  Default priority of the thread.
- uint8_t m_u8CurPriority

  Current priority of the thread (priority inheritence)
- ThreadState_t m_eState

  Enum indicating the thread's current state.

14.8.1 Detailed Description

If the kernel is set up to use an idle function instead of an idle thread, we use a placeholder data structure to "simulate" the effect of having an idle thread in the system.

When cast to a Thread, this data structure will still result in GetPriority() calls being valid, which is all that is needed to support the tick-based/tickless times – while saving a fairly decent chunk of RAM on a small micro.

Note that this struct must have the same memory layout as the Thread class up to the last item.

Definition at line 494 of file thread.h.

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

- /home/vm/mark3/trunk/embedded/kernel/public/thread.h

14.9 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 popu32ated 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.

14.9.1 Detailed Description

Implements a list of message objects shared between all threads.

Definition at line 157 of file message.h.

14.9.2 Member Function Documentation

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

Pop a message from the global queue, returning it to the user to be popu32ated before sending by a transmitter.

Returns

Pointer to a Message object

Definition at line 74 of file message.cpp.

14.9.2.2 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 62 of file message.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/public/message.h
- /home/vm/mark3/trunk/embedded/kernel/message.cpp

14.10 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 (uint16_t u16Cause_)

  Panic Cause the kernel to enter its panic state.
- static void SetIdleFunc (idle_func_t pfIdle_)

  SetIdleFunc Set the function to be called when no active threads are available to be scheduled by the scheduler.
- static void IdleFunc (void)

  IdleFunc Call the low-priority idle function when no active threads are available to be scheduled.
- static Thread * GetIdleThread (void)

  GetIdleThread Return a pointer to the Kernel's idle thread object to the user.

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

  set panic function
- static idle_func_t m_pfIdle

set idle function

- static FakeThread_t m_clIdle

  Idle thread object (note: not a real thread)

14.10.1 Detailed Description

Class that encapsulates all of the kernel startup functions.

Definition at line 48 of file kernel.h.

14.10.2 Member Function Documentation

14.10.2.1 static Thread* Kernel::GetIdleThread (void) [inline], [static]

GetIdleThread Return a pointer to the Kernel's idle thread object to the user.

Note that the Thread object involved is to be used for comparisons only – the thread itself is "virtual", and doesn't represent a unique execution context with its own stack.

Returns

Pointer to the Kernel's idle thread object

Definition at line 124 of file kernel.h.

14.10.2.2 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 57 of file kernel.cpp.

14.10.2.3 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 95 of file kernel.h.

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

IsStarted.

Returns

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

Definition at line 80 of file kernel.h.

14.10.2.5 void Kernel::Panic (uint16_t *u16Cause_*) [static]

Panic Cause the kernel to enter its panic state.

Parameters

*u16Cause_*	Reason for the kernel panic

Definition at line 100 of file kernel.cpp.

14.10.2.6 static void Kernel::SetIdleFunc (idle_func_t *pfIdle_*) [inline], [static]

SetIdleFunc Set the function to be called when no active threads are available to be scheduled by the scheduler.

Parameters

*pfIdle_*	Pointer to the idle function

Definition at line 109 of file kernel.h.

14.10.2.7 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 89 of file kernel.h.

14.10.2.8 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 91 of file kernel.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/public/kernel.h
- /home/vm/mark3/trunk/embedded/kernel/kernel.cpp

14.11 KernelAware Class Reference

The KernelAware class.

```
#include <kernelaware.h>
```

Static Public Member Functions

- static void ProfileInit (const char *szStr_)

  ProfileInit.
- static void ProfileStart (void)

  ProfileStart.
- static void ProfileStop (void)

  ProfileStop.

- static void ProfileReport (void)

  ProfileReport.
- static void ExitSimulator (void)

  ExitSimulator.
- static void Print (const char *szStr_)

  Print.
- static void Trace (uint16_t u16File_, uint16_t u16Line_)

  Trace.
- static void Trace (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_)

  Trace.
- static void Trace (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_)

  Trace.
- static bool IsSimulatorAware (void)

  IsSimulatorAware.

Static Private Member Functions

- static void Trace_i (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_, KernelAwareCommand_t eCmd_)

  Trace_i.

14.11.1 Detailed Description

The KernelAware class.

This class contains functions that are used to trigger kernel-aware functionality within a supported simulation environment (i.e. flAVR).

These static methods operate on a singleton set of global variables, which are monitored for changes from within the simulator. The simulator hooks into these variables by looking for the correctly-named symbols in an elf-formatted binary being run and registering callbacks that are called whenever the variables are changed. On each change of the command variable, the kernel-aware data is analyzed and interpreted appropriately.

If these methods are run in an unsupported simulator or on actual hardware the commands generally have no effect (except for the exit-on-reset command, which will result in a jump-to-0 reset).

Definition at line 65 of file kernelaware.h.

14.11.2 Member Function Documentation

14.11.2.1 void KernelAware::ExitSimulator (void) [static]

ExitSimulator.

Instruct the kernel-aware simulator to terminate (destroying the virtual CPU).

Definition at line 114 of file kernelaware.cpp.

14.11.2.2 bool KernelAware::IsSimulatorAware (void) [static]

IsSimulatorAware.

use this function to determine whether or not the code is running on a simulator that is aware of the kernel.

Returns

true - the application is being run in a kernel-aware simulator. false - otherwise.

Definition at line 169 of file kernelaware.cpp.

14.11.2.3 void KernelAware::Print (const char * *szStr_*) [static]

Print.

Instruct the kernel-aware simulator to print a char string

Parameters

*szStr_*	

Definition at line 160 of file kernelaware.cpp.

14.11.2.4 void KernelAware::ProfileInit (const char * *szStr_*) [static]

ProfileInit.

Initializes the kernel-aware profiler. This function instructs the kernel-aware simulator to reset its accounting variables, and prepare to start counting profiling data tagged to the given string. How this is handled is the responsibility of the simulator.

Parameters

*szStr_*	String to use as a tag for the profilng session.

Definition at line 87 of file kernelaware.cpp.

14.11.2.5 void KernelAware::ProfileReport (void) [static]

ProfileReport.

Instruct the kernel-aware simulator to print a report for its current profiling data.

Definition at line 108 of file kernelaware.cpp.

14.11.2.6 void KernelAware::ProfileStart (void) [static]

ProfileStart.

Instruct the kernel-aware simulator to begin counting cycles towards the current profiling counter.

Definition at line 96 of file kernelaware.cpp.

14.11.2.7 void KernelAware::ProfileStop (void) [static]

ProfileStop.

Instruct the kernel-aware simulator to end counting cycles relative to the current profiling counter's iteration.

Definition at line 102 of file kernelaware.cpp.

14.11.2.8 void KernelAware::Trace (uint16_t *u16File_*, uint16_t *u16Line_*) [static]

Trace.

Insert a kernel trace statement into the kernel-aware simulator's debug data stream.

Parameters

*u16File_*	16-bit code representing the file
*u16Line_*	16-bit code representing the line in the file

Definition at line 120 of file kernelaware.cpp.

14.11.2.9 void KernelAware::Trace (uint16_t *u16File_*, uint16_t *u16Line_*, uint16_t *u16Arg1_*) [static]

Trace.

Insert a kernel trace statement into the kernel-aware simulator's debug data stream.

Parameters

*u16File_*	16-bit code representing the file
*u16Line_*	16-bit code representing the line in the file
*u16Arg1_*	16-bit argument to the format string.

Definition at line 127 of file kernelaware.cpp.

14.11.2.10 void KernelAware::Trace (uint16_t *u16File_*, uint16_t *u16Line_*, uint16_t *u16Arg1_*, uint16_t *u16Arg2_*) [static]

Trace.

Insert a kernel trace statement into the kernel-aware simulator's debug data stream.

Parameters

*u16File_*	16-bit code representing the file
*u16Line_*	16-bit code representing the line in the file
*u16Arg1_*	16-bit argument to the format string.
*u16Arg2_*	16-bit argument to the format string.

Definition at line 135 of file kernelaware.cpp.

14.11.2.11 void KernelAware::Trace_i (uint16_t *u16File_*, uint16_t *u16Line_*, uint16_t *u16Arg1_*, uint16_t *u16Arg2_*, KernelAwareCommand_t *eCmd_*) [static], [private]

Trace_i.

Private function by which the class's Trace() methods are reflected, which allows u16 to realize a modest code saving.

Parameters

*u16File_*	16-bit code representing the file
*u16Line_*	16-bit code representing the line in the file
*u16Arg1_*	16-bit argument to the format string.
*u16Arg2_*	16-bit argument to the format string.
*eCmd_*	Code indicating the number of arguments to emit.

Definition at line 144 of file kernelaware.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/public/kernelaware.h
- /home/vm/mark3/trunk/embedded/kernel/kernelaware.cpp

14.12 KernelAwareData_t Union Reference

This structure is used to communicate between the kernel and a kernel- aware host.

Public Attributes

- volatile uint16_t au16Buffer [5]

  Raw binary contents of the struct.

- *The Profiler struct contains data related to the code-execution profiling functionality provided by a kernel-aware host simlua-tor.*

  The Trace struct contains data related to the display and output of kernel-trace strings on a kernel-aware host.

  The Print struct contains data related to the display of arbitrary null-terminated ASCII strings on the kernel-aware host.

14.12.1 Detailed Description

This structure is used to communicate between the kernel and a kernel- aware host.

Its data contents is interpreted differently depending on the command executed (by means of setting the g_u8KA↩ Command variable, as is done in the command handlers in this module). As a result, any changes to this struct by way of modifying or adding data must be mirrored in the kernel-aware simulator.

Definition at line 48 of file kernelaware.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/kernelaware.cpp

14.13 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 uint8_t DI ()

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

  Restore the state of the SWI to the value specified.

14.13.1 Detailed Description

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

Definition at line 32 of file kernelswi.h.

14.13.2 Member Function Documentation

14.13.2.1 uint8_t 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.

14.13.2.2 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:

- /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h
- /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp

14.14 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 uint8_t 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 uint32_t SubtractExpiry (uint32_t u32Interval_)

  Subtract the specified number of ticks from the timer's expiry count register.
- static uint32_t TimeToExpiry (void)

  Returns the number of ticks remaining before the next timer expiry.
- static uint32_t SetExpiry (uint32_t u32Interval_)

  Resets the kernel timer's expiry interval to the specified value.
- static uint32_t 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 uint16_t Read (void)

  Safely read the current value in the timer register.

14.14.1 Detailed Description

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

Definition at line 33 of file kerneltimer.h.

14.14.2 Member Function Documentation

14.14.2.1 uint32_t 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.

14.14.2.2 uint16_t 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.

14.14.2.3 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 169 of file kerneltimer.cpp.

14.14.2.4 uint32_t KernelTimer::SetExpiry (uint32_t *u32Interval_*) [static]

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

Parameters

*u32Interval_*	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.

14.14.2.5 uint32_t KernelTimer::SubtractExpiry (uint32_t *u32Interval_*) [static]

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

Returns the new expiry value stored in the register.

Parameters

*u32Interval_*	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.

14.14.2.6 uint32_t 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:

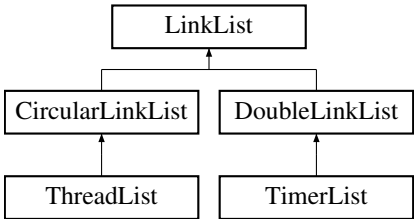
- /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h
- /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/kerneltimer.cpp

14.15 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.

14.15.1 Detailed Description

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

Definition at line 112 of file ll.h.

14.15.2 Member Function Documentation

14.15.2.1 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.

14.15.2.2 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.

14.15.2.3 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.

14.15.2.4 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:

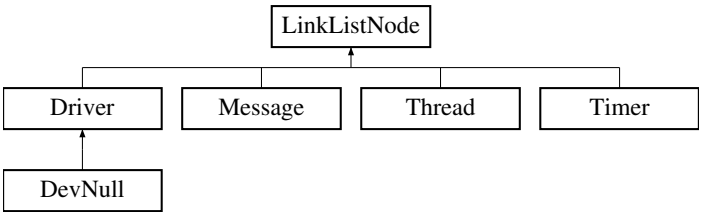
- /home/vm/mark3/trunk/embedded/kernel/public/ll.h

14.16 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**

14.16.1 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.

14.16.2 Member Function Documentation

14.16.2.1 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.

14.16.2.2 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:

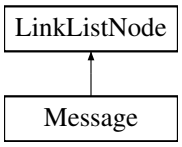
- /home/vm/mark3/trunk/embedded/kernel/public/ll.h
- /home/vm/mark3/trunk/embedded/kernel/ll.cpp

14.17 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 (uint16_t u16Code_)
  Set the code in the message before transmission.
- uint16_t GetCode ()
  Return the code set in the message upon receipt.

Private Attributes

- void * m_pvData
  Pointer to the message data.
- uint16_t m_u16Code
  Message code, providing context for the message.

Additional Inherited Members

14.17.1 Detailed Description

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

Definition at line 99 of file message.h.

14.17.2 Member Function Documentation

14.17.2.1 uint16_t 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.

14.17.2.2 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.

14.17.2.3 Message::SetCode (uint16_t *u16Code_*) [inline]

Set the code in the message before transmission.

Parameters

*u16Code_*	Data code to set in the object

Definition at line 134 of file message.h.

14.17.2.4 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:

- /home/vm/mark3/trunk/embedded/kernel/public/message.h

14.18 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.
- Message * Receive (uint32_t u32TimeWaitMS_)

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

  Send a message object into this message queue.
- uint16_t GetCount ()

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

Private Member Functions

- Message * Receive_i (uint32_t u32TimeWaitMS_)

  Receive_i.

Private Attributes

- Semaphore m_clSemaphore

  Counting semaphore used to manage thread blocking.
- DoubleLinkList m_clLinkList

  List object used to store messages.

14.18.1 Detailed Description

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

Definition at line 201 of file message.h.

14.18.2 Member Function Documentation

14.18.2.1 uint16_t MessageQueue::GetCount ( )

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

Returns

Count of pending messages in the queue.

Definition at line 160 of file message.cpp.

14.18.2.2 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 96 of file message.cpp.

14.18.2.3 Message * MessageQueue::Receive (uint32_t *u32WaitTimeMS_*)

Receive a message from the message queue.

If the message queue is empty, the thread will block until a message is available for the duration specified. If no message arrives within that duration, the call will return with NULL.

Parameters

*u32WaitTimeMS_*	The amount of time in ms to wait for a message before timing out and unblocking the waiting thread.

Returns

Pointer to a message object at the head of the queue or NULL on timeout.

Definition at line 107 of file message.cpp.

14.18.2.4 Message * MessageQueue::Receive_i (uint32_t *u32TimeWaitMS_*) [private]

Receive_i.

Internal function used to abstract timed and un-timed Receive calls.

Parameters

*u32TimeWaitMS_*	Time (in ms) to block, 0 for un-timed call.

Returns

Pointer to a message, or 0 on timeout.

Definition at line 115 of file message.cpp.

14.18.2.5 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 144 of file message.cpp.

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

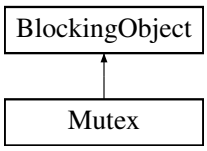
- /home/vm/mark3/trunk/embedded/kernel/public/message.h
- /home/vm/mark3/trunk/embedded/kernel/message.cpp

14.19 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.
- bool Claim (uint32_t u32WaitTimeMS_)
- void WakeMe (Thread *pclOwner_)

  Wake a thread blocked on the mutex.
- void Release ()

  Release the mutex.

Private Member Functions

- uint8_t WakeNext ()

  Wake the next thread waiting on the Mutex.
- bool Claim_i (uint32_t u32WaitTimeMS_)

  Claim_i.

Private Attributes

- uint8_t m_u8Recurse

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

  State of the mutex - true = ready, false = claimed.
- uint8_t m_u8MaxPri

  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

14.19.1 Detailed Description

Mutual-exclusion locks, based on BlockingObject.

Definition at line 68 of file mutex.h.

14.19.2 Member Function Documentation

14.19.2.1 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 215 of file mutex.cpp.

14.19.2.2 bool Mutex::Claim (uint32_t *u32WaitTimeMS_*)

Parameters

*u32WaitTimeMS_*	

Returns

true - mutex was claimed within the time period specified false - mutex operation timed-out before the claim operation.

Definition at line 226 of file mutex.cpp.

14.19.2.3 bool Mutex::Claim_i (uint32_t *u32WaitTimeMS_*) [private]

Claim_i.

Abstracts out timed/non-timed mutex claim operations.

Parameters

*u32WaitTimeMS_*	Time in MS to wait, 0 for infinite

Returns

true on successful claim, false otherwise

Definition at line 113 of file mutex.cpp.

14.19.2.4 void Mutex::Release ( )

Release the mutex.

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

Definition at line 233 of file mutex.cpp.

14.19.2.5 void Mutex::WakeMe (Thread * *pclOwner_*)

Wake a thread blocked on the mutex.

This is an internal function used for implementing timed mutexes relying on timer callbacks. Since these do not have access to the private data of the mutex and its base classes, we have to wrap this as a public method - do not use this for any other purposes.

Parameters

*pclOwner_*	Thread to unblock from this object.

Definition at line 71 of file mutex.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/public/mutex.h
- /home/vm/mark3/trunk/embedded/kernel/mutex.cpp

14.20 Profiler Class Reference

System profiling timer interface.

`#include <kernelprofile.h>`

Static Public Member Functions

- static void Init ()
  Initialize the global system profiler.
- static void Start ()
  Start the global profiling timer service.
- static void Stop ()
  Stop the global profiling timer service.
- static uint16_t Read ()
  Read the current tick count in the timer.
- static void Process ()
  Process the profiling counters from ISR.
- static uint32_t GetEpoch ()
  Return the current timer epoch.

14.20.1 Detailed Description

System profiling timer interface.

Definition at line 37 of file kernelprofile.h.

14.20.2 Member Function Documentation

14.20.2.1 void Profiler::Init (void) [static]

Initialize the global system profiler.

Must be called prior to use.

Definition at line 32 of file kernelprofile.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h
- /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp

14.21 ProfileTimer Class Reference

Profiling timer.

`#include <profile.h>`

Public Member Functions

- void Init ()

  Initialize the profiling timer prior to use.
- void Start ()

  Start a profiling session, if the timer is not already active.
- void Stop ()

  Stop the current profiling session, adding to the cumulative time for this timer, and the total iteration count.
- uint32_t GetAverage ()

  Get the average time associated with this operation.
- uint32_t GetCurrent ()

  Return the current tick count held by the profiler.

Private Member Functions

- uint32_t ComputeCurrentTicks (uint16_t u16Count_, uint32_t u32Epoch_)

  Figure out how many ticks have elapsed in this iteration.

Private Attributes

- uint32_t m_u32Cumulative

  Cumulative tick-count for this timer.
- uint32_t m_u32CurrentIteration

  Tick-count for the current iteration.
- uint16_t m_u16Initial

  Initial count.

- uint32_t m_u32InitialEpoch

  Initial Epoch.
- uint16_t m_u16Iterations

  Number of iterations executed for this profiling timer.
- bool m_bActive

  Wheter or not the timer is active or stopped.

14.21.1 Detailed Description

Profiling timer.

This class is used to perform high-performance profiling of code to see how int32_t certain operations take. useful in instrumenting the performance of key algorithms and time-critical operations to ensure real-timer behavior.

Definition at line 69 of file profile.h.

14.21.2 Member Function Documentation

14.21.2.1 uint32_t ProfileTimer::ComputeCurrentTicks (uint16_t *u16Count_*, uint32_t *u32Epoch_*) [private]

Figure out how many ticks have elapsed in this iteration.

Parameters

*u16Count_*	Current timer count
*u32Epoch_*	Current timer epoch

Returns

Current tick count

Definition at line 112 of file profile.cpp.

14.21.2.2 uint32_t ProfileTimer::GetAverage ( )

Get the average time associated with this operation.

Returns

Average tick count normalized over all iterations

Definition at line 85 of file profile.cpp.

14.21.2.3 uint32_t ProfileTimer::GetCurrent ( )

Return the current tick count held by the profiler.

Valid for both active and stopped timers.

Returns

The currently held tick count.

Definition at line 95 of file profile.cpp.

14.21.2.4 void ProfileTimer::Init (void)

Initialize the profiling timer prior to use.

Can also be used to reset a timer that's been used previously.

Definition at line 43 of file profile.cpp.

14.21.2.5 void ProfileTimer::Start (void)

Start a profiling session, if the timer is not already active.

Has no effect if the timer is already active.

Definition at line 52 of file profile.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/public/profile.h
- /home/vm/mark3/trunk/embedded/kernel/profile.cpp

14.22 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.

14.22.1 Detailed Description

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

Definition at line 41 of file quantum.h.

14.22.2 Member Function Documentation

14.22.2.1 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 88 of file quantum.cpp.

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

ClearInTimer.

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

Definition at line 84 of file quantum.h.

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

Remove the thread from the quantum timer.

This will cancel the timer.

Definition at line 117 of file quantum.cpp.

14.22.2.4 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 77 of file quantum.h.

14.22.2.5 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 78 of file quantum.cpp.

14.22.2.6 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 130 of file quantum.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/public/quantum.h
- /home/vm/mark3/trunk/embedded/kernel/quantum.cpp

14.23 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 bool SetScheduler (bool bEnable_)

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

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

  Return the pointer to the thread that should run next, according to the last run of the scheduler.
- static ThreadList * GetThreadList (uint8_t u8Priority_)

  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 uint8_t IsEnabled ()

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

  QueueScheduler.

Static Private Attributes

- static bool m_bEnabled

  Scheduler's state - enabled or disabled.
- static 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 uint8_t m_u8PriFlag

  Bitmap flag for each.

14.23.1 Detailed Description

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

Definition at line 62 of file scheduler.h.

14.23.2 Member Function Documentation

14.23.2.1 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 113 of file scheduler.cpp.

14.23.2.2 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.

14.23.2.3 static volatile 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.

14.23.2.4 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.

14.23.2.5 static ThreadList* Scheduler::GetThreadList (uint8_t *u8Priority_*) [inline], [static]

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

Parameters

*u8Priority_*	Priority level of

Returns

Pointer to the ThreadList for the given priority level

Definition at line 137 of file scheduler.h.

14.23.2.6 uint8_t 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.

14.23.2.7 static void Scheduler::QueueScheduler ( ) [inline], [static]

QueueScheduler.

Tell the kernel to perform a scheduling operation as soon as the scheduler is re-enabled.

Definition at line 163 of file scheduler.h.

14.23.2.8 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 119 of file scheduler.cpp.

14.23.2.9 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 76 of file scheduler.cpp.

14.23.2.10 void Scheduler::SetScheduler (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 125 of file scheduler.cpp.

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

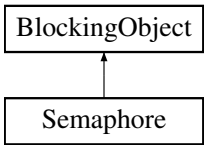
- /home/vm/mark3/trunk/embedded/kernel/public/scheduler.h
- /home/vm/mark3/trunk/embedded/kernel/scheduler.cpp

14.24 Semaphore Class Reference

Counting semaphore, based on BlockingObject base class.

`#include <ksemaphore.h>`

Inheritance diagram for Semaphore:



Public Member Functions

- void Init (uint16_t u16InitVal_, uint16_t u16MaxVal_)

  Initialize a semaphore before use.
- bool Post ()

  Increment the semaphore count.
- void Pend ()

  Decrement the semaphore count.
- uint16_t GetCount ()

  Return the current semaphore counter.
- bool Pend (uint32_t u32WaitTimeMS_)

  Decrement the semaphore count.
- void WakeMe (Thread *pclChosenOne_)

  Wake a thread blocked on the semaphore.

Private Member Functions

- uint8_t WakeNext ()

  Wake the next thread waiting on the semaphore.
- bool Pend_i (uint32_t u32WaitTimeMS_)

  Pend_i.

Private Attributes

- uint16_t m_u16Value

  Current count held by the semaphore.
- uint16_t m_u16MaxValue

  Maximum count that can be held by this semaphore.

Additional Inherited Members

14.24.1 Detailed Description

Counting semaphore, based on BlockingObject base class.

Definition at line 37 of file ksemaphore.h.

14.24.2 Member Function Documentation

14.24.2.1 uint16_t Semaphore::GetCount ( )

Return the current semaphore counter.

This can be usedd 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 241 of file ksemaphore.cpp.

14.24.2.2 void Semaphore::Init (uint16_t *u16InitVal_*, uint16_t *u16MaxVal_*)

Initialize a semaphore before use.

Must be called before post/pend operations.

Parameters

*u16InitVal_*	Initial value held by the semaphore
*u16MaxVal_*	Maximum value for the semaphore

Definition at line 102 of file ksemaphore.cpp.

14.24.2.3 void Semaphore::Pend ( )

Decrement the semaphore count.

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

Definition at line 223 of file ksemaphore.cpp.

14.24.2.4 bool Semaphore::Pend (uint32_t *u32WaitTimeMS_*)

Decrement the semaphore count.

If the count is zero, the thread will block until the semaphore is pended. If the specified interval expires before the thread is unblocked, then the status is returned back to the user.

Returns

true - semaphore was acquired before the timeout false - timeout occurred before the semaphore was claimed.

Definition at line 234 of file ksemaphore.cpp.

14.24.2.5 bool Semaphore::Pend_i (uint32_t *u32WaitTimeMS_*) [private]

Pend_i.

Internal function used to abstract timed and untimed semaphore pend operations.

Parameters

*u32WaitTimeMS_*	Time in MS to wait

Returns

true on success, false on failure.

Definition at line 167 of file ksemaphore.cpp.

14.24.2.6 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 114 of file ksemaphore.cpp.

14.24.2.7 void Semaphore::WakeMe (Thread * *pclChosenOne_*)

Wake a thread blocked on the semaphore.

This is an internal function used for implementing timed semaphores relying on timer callbacks. Since these do not have access to the private data of the semaphore and its base classes, we have to wrap this as a public method - do not used this for any other purposes.

Definition at line 75 of file ksemaphore.cpp.

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

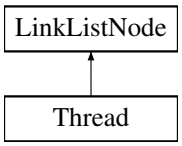
- /home/vm/mark3/trunk/embedded/kernel/public/ksemaphore.h
- /home/vm/mark3/trunk/embedded/kernel/ksemaphore.cpp

14.25 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_, uint16_t u16StackSize_, uint8_t u8Priority_, 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.
- 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.
- uint8_t GetPriority (void)

  Return the priority of the current thread.
- uint8_t GetCurPriority (void)

  Return the priority of the current thread.
- void SetQuantum (uint16_t u16Quantum_)

  Set the thread's round-robin execution quantum.
- uint16_t 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 (uint8_t u8Priority_)

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

- void InheritPriority (uint8_t u8Priority_)

  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 (uint8_t u8ID_)

  Set an 8-bit ID to uniquely identify this thread.
- uint8_t GetID ()

  Return the 8-bit ID corresponding to this thread.
- uint16_t GetStackSlack ()

  Performs a (somewhat lengthy) check on the thread stack to check the amount of stack margin (or "slack") remaining on the stack.
- uint16_t GetEventFlagMask ()

  GetEventFlagMask returns the thread's current event-flag mask, which is used in conjunction with the EventFlag blocking object type.
- void SetEventFlagMask (uint16_t u16Mask_)

  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 (bool bExpired_)

  SetExpired.
- bool GetExpired ()

  GetExpired.
- void InitIdle ()

  InitIdle Initialize this Thread object as the Kernel's idle thread.
- ThreadState_t GetState ()

  GetState Returns the current state of the thread to the caller.
- void SetState (ThreadState_t eState_)

  SetState Set the thread's state to a new value.

Static Public Member Functions

- static void Sleep (uint32_t u32TimeMs_)

  Put the thread to sleep for the specified time (in milliseconds).
- static void USleep (uint32_t u32TimeUs_)

  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 (uint8_t u8Priority_)

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.
- uint8_t m_u8ThreadID

  Thread ID.
- uint8_t m_u8Priority

  Default priority of the thread.
- uint8_t m_u8CurPriority

  Current priority of the thread (priority inheritence)
- ThreadState_t m_eState

  Enum indicating the thread's current state.
- uint16_t m_u16StackSize

  Size of the stack (in bytes)
- 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.
- 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.
- uint16_t m_u16Quantum

  Thread quantum (in milliseconds)
- uint16_t m_u16FlagMask

  Event-flag mask.
- EventFlagOperation_t m_eFlagMode

  Event-flag mode.
- Timer m_clTimer

  Timer used for blocking-object timeouts.
- bool m_bExpired

  Indicate whether or not a blocking-object timeout has occurred.

Friends

- class **ThreadPort**

Additional Inherited Members

14.25.1 Detailed Description

Object providing fundamental multitasking support in the kernel.

Definition at line 71 of file thread.h.

14.25.2 Member Function Documentation

14.25.2.1 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 400 of file thread.cpp.

14.25.2.2 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 179 of file thread.cpp.

14.25.2.3 uint8_t Thread::GetCurPriority (void) [inline]

Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 174 of file thread.h.

14.25.2.4 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 155 of file thread.h.

14.25.2.5 uint16_t 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 327 of file thread.h.

14.25.2.6 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 346 of file thread.h.

14.25.2.7 bool Thread::GetExpired ( )

GetExpired.

Return the status of the most-recent blocking call on the thread.

Returns

true - call expired, false - call did not expire

Definition at line 418 of file thread.cpp.

14.25.2.8 uint8_t 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 302 of file thread.h.

14.25.2.9 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 146 of file thread.h.

14.25.2.10 uint8_t Thread::GetPriority (void) [inline]

Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 165 of file thread.h.

14.25.2.11 uint16_t Thread::GetQuantum (void) [inline]

Get the thread's round-robin execution quantum.

Returns

The thread's quantum

Definition at line 193 of file thread.h.

14.25.2.12 uint16_t 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 289 of file thread.cpp.

14.25.2.13 ThreadState_t Thread::GetState ( ) [inline]

GetState Returns the current state of the thread to the caller.

Can be used to determine whether or not a thread is ready (or running), stopped, or terminated/exit'd.

Returns

ThreadState_t representing the thread's current state

Definition at line 390 of file thread.h.

14.25.2.14 void Thread::InheritPriority (uint8_t *u8Priority_*)

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

*u8Priority_*	New Priority to boost to.

Definition at line 393 of file thread.cpp.

14.25.2.15 void Thread::Init (K_WORD * *paucStack_*, uint16_t *u16StackSize_*, uint8_t *u8Priority_*, 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
*u16StackSize_*	Size of the stack (in bytes)
*u8Priority_*	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.

Definition at line 46 of file thread.cpp.

14.25.2.16 void Thread::InitIdle (void)

InitIdle Initialize this Thread object as the Kernel's idle thread.

There should only be one of these, maximum, in a given system.

Definition at line 423 of file thread.cpp.

14.25.2.17 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 203 of file thread.h.

14.25.2.18 void Thread::SetEventFlagMask (uint16_t *u16Mask_*) [inline]

SetEventFlagMask Sets the active event flag bitfield mask.

Parameters

*u16Mask_*	

Definition at line 333 of file thread.h.

14.25.2.19 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 340 of file thread.h.

14.25.2.20 void Thread::SetExpired (bool *bExpired_*)

SetExpired.

Set the status of the current blocking call on the thread.

Parameters

*bExpired_*	true - call expired, false - call did not expire

Definition at line 415 of file thread.cpp.

14.25.2.21 void Thread::SetID (uint8_t *u8ID_*) [inline]

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

Parameters

*u8ID_*	8-bit Thread ID, set by the user

Definition at line 293 of file thread.h.

14.25.2.22 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 212 of file thread.h.

14.25.2.23 void Thread::SetPriority (uint8_t *u8Priority_*)

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

*u8Priority_*	New priority of the thread

Definition at line 349 of file thread.cpp.

14.25.2.24 void Thread::SetPriorityBase (uint8_t *u8Priority_*) [private]

Parameters

*u8Priority_*	

Definition at line 339 of file thread.cpp.

14.25.2.25 void Thread::SetQuantum (uint16_t *u16Quantum_*) [inline]

Set the thread's round-robin execution quantum.

Parameters

*u16Quantum_*	Thread's execution quantum (in milliseconds)

Definition at line 184 of file thread.h.

14.25.2.26 void Thread::SetState (ThreadState_t *eState_*) [inline]

SetState Set the thread's state to a new value.

This is only to be used by code within the kernel, and is not indended for use by an end-user.

Parameters

*eState_*	New thread state to set.

Definition at line 399 of file thread.h.

14.25.2.27 void Thread::Sleep (uint32_t *u32TimeMs_*) [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

*u32TimeMs_*	Time to sleep (in ms)

Definition at line 244 of file thread.cpp.

14.25.2.28 void Thread::Stop ( )

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

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

Definition at line 134 of file thread.cpp.

14.25.2.29 void Thread::USleep (uint32_t *u32TimeUs_*) [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

*u32TimeUs_*	Time to sleep (in microseconds)

Definition at line 266 of file thread.cpp.

14.25.2.30 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 310 of file thread.cpp.

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

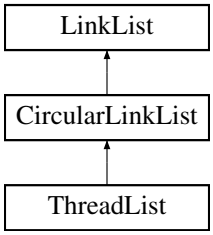
- /home/vm/mark3/trunk/embedded/kernel/public/thread.h
- /home/vm/mark3/trunk/embedded/kernel/thread.cpp

14.26 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 (uint8_t u8Priority_)

  Set the priority of this threadlist (if used for a scheduler).
- void SetFlagPointer (uint8_t *pu8Flag_)

  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_, uint8_t *pu8Flag_, uint8_t u8Priority_)

  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

- uint8_t m_u8Priority

  Priority of the threadlist.
- uint8_t * m_pu8Flag

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

Additional Inherited Members

14.26.1 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.

14.26.2 Member Function Documentation

14.26.2.1 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 52 of file threadlist.cpp.

14.26.2.2 void ThreadList::Add (LinkListNode * *node_*, uint8_t * *pu8Flag_*, uint8_t *u8Priority_*)

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)
*pu8Flag_*	Pointer to the bitmap flag to set (if used in a scheduler context), or NULL for non-scheduler.
*u8Priority_*	Priority of the threadlist

Definition at line 65 of file threadlist.cpp.

14.26.2.3 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 90 of file threadlist.cpp.

14.26.2.4 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 74 of file threadlist.cpp.

14.26.2.5 void ThreadList::SetFlagPointer (uint8_t * *pu8Flag_*)

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

*pu8Flag_*	Pointer to the bitmap flag

Definition at line 46 of file threadlist.cpp.

14.26.2.6 void ThreadList::SetPriority (uint8_t *u8Priority_*)

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

Parameters

*u8Priority_*	Priority level of the thread list

Definition at line 40 of file threadlist.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/public/threadlist.h
- /home/vm/mark3/trunk/embedded/kernel/threadlist.cpp

14.27 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**

14.27.1 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.

14.27.2 Member Function Documentation

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

Initialize the thread's stack.

Parameters

*pstThread_*	Pointer to the thread to initialize

Definition at line 39 of file threadport.cpp.

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

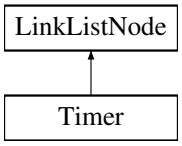
- /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/public/threadport.h
- /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/threadport.cpp

14.28 Timer Class Reference

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

`#include <timer.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_, uint32_t u32IntervalMs_, TimerCallback_t pfCallback_, void *pvData_)

  Start a timer using default ownership, using repeats as an option, and millisecond resolution.
- void Start (bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_, TimerCallback_t pf↩ Callback_, 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 (uint8_t u8Flags_)

  Set the timer's flags based on the bits in the u8Flags_ 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 (uint32_t u32Ticks_)

  Set the timer expiry in system-ticks (platform specific!)
- void SetIntervalSeconds (uint32_t u32Seconds_)

  ! The next three cost u16 330 bytes of flash on AVR...
- void SetIntervalMSeconds (uint32_t u32MSeconds_)

  Set the timer expiry interval in milliseconds (platform agnostic)
- void SetIntervalUSeconds (uint32_t u32USeconds_)

  Set the timer expiry interval in microseconds (platform agnostic)
- void SetTolerance (uint32_t u32Ticks_)

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

Private Attributes

- uint8_t m_u8Flags

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

  Pointer to the callback function.
- uint32_t m_u32Interval

  Interval of the timer in timer ticks.
- uint32_t m_u32TimeLeft

  Time remaining on the timer.
- uint32_t m_u32TimerTolerance

  Maximum tolerance (usedd 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

14.28.1 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 102 of file timer.h.

14.28.2 Member Function Documentation

14.28.2.1 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 163 of file timer.h.

14.28.2.2 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 172 of file timer.h.

14.28.2.3 void Timer::SetFlags (uint8_t *u8Flags_*) [inline]

Set the timer's flags based on the bits in the u8Flags_ argument.

Parameters

*u8Flags_*	Flags to assign to the timer object. TIMERLIST_FLAG_ONE_SHOT for a one-shot timer, 0 for a continuous timer.

Definition at line 154 of file timer.h.

14.28.2.4 void Timer::SetIntervalMSeconds (uint32_t *u32MSeconds_*)

Set the timer expiry interval in milliseconds (platform agnostic)

Parameters

*u32MSeconds_*	Time in milliseconds

Definition at line 94 of file timer.cpp.

14.28.2.5 void Timer::SetIntervalSeconds (uint32_t *u32Seconds_*)

! The next three cost u16 330 bytes of flash on AVR...

Set the timer expiry interval in seconds (platform agnostic)

Parameters

*u32Seconds_*	Time in seconds

Definition at line 88 of file timer.cpp.

14.28.2.6 void Timer::SetIntervalTicks (uint32_t *u32Ticks_*)

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

Parameters

*u32Ticks_*	Time in ticks

Definition at line 80 of file timer.cpp.

14.28.2.7 void Timer::SetIntervalUSeconds (uint32_t *u32USeconds_*)

Set the timer expiry interval in microseconds (platform agnostic)

Parameters

*u32USeconds_*	Time in microseconds

Definition at line 100 of file timer.cpp.

14.28.2.8 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 182 of file timer.h.

14.28.2.9 void Timer::SetTolerance (uint32_t *u32Ticks_*)

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

Parameters

*u32Ticks_*	Maximum tolerance in ticks

Definition at line 106 of file timer.cpp.

14.28.2.10 void Timer::Start (bool *bRepeat_*, uint32_t *u32IntervalMs_*, 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.
*u32IntervalMs_*	- Interval of the timer in miliseconds

*pfCallback_*	- Function to call on timer expiry
*pvData_*	- Data to pass into the callback function

Definition at line 48 of file timer.cpp.

14.28.2.11 void Timer::Start (bool *bRepeat_*, uint32_t *u32IntervalMs_*, uint32_t *u32ToleranceMs_*, 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.
*u32IntervalMs_*	- Interval of the timer in miliseconds
u32ToleranceMs	- 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 67 of file timer.cpp.

14.28.2.12 void Timer::Stop ( )

Stop a timer already in progress.

Has no effect on timers that have already been stopped.

Definition at line 74 of file timer.cpp.

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

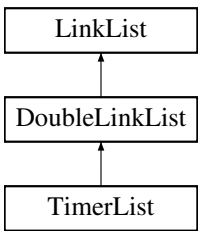
- /home/vm/mark3/trunk/embedded/kernel/public/timer.h
- /home/vm/mark3/trunk/embedded/kernel/timer.cpp

14.29 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

- uint32_t m_u32NextWakeup

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

- bool m_bTimerActive

  Whether or not the timer is active.

Additional Inherited Members

14.29.1 Detailed Description

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

Definition at line 37 of file timerlist.h.

14.29.2 Member Function Documentation

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

Add a timer to the TimerList.

Parameters

*pclListNode_*	Pointer to the Timer to Add

Definition at line 56 of file timerlist.cpp.

14.29.2.2 void TimerList::Init (void)

Initialize the TimerList object.

Must be called before using the object.

Definition at line 49 of file timerlist.cpp.

14.29.2.3 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 121 of file timerlist.cpp.

14.29.2.4 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 104 of file timerlist.cpp.

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

- /home/vm/mark3/trunk/embedded/kernel/public/timerlist.h
- /home/vm/mark3/trunk/embedded/kernel/timerlist.cpp

14.30 TimerScheduler Class Reference

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

`#include <timerscheduler.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 manipu32ated by the Timer Scheduler.

14.30.1 Detailed Description

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

Definition at line 38 of file timerscheduler.h.

14.30.2 Member Function Documentation

14.30.2.1 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 57 of file timerscheduler.h.

14.30.2.2 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 47 of file timerscheduler.h.

14.30.2.3 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. The next timer epoch is set based on the next Timer object to expire.

Definition at line 79 of file timerscheduler.h.

14.30.2.4 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 68 of file timerscheduler.h.

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

- /home/vm/mark3/trunk/embedded/kernel/public/timerscheduler.h
- /home/vm/mark3/trunk/embedded/kernel/timerlist.cpp

Chapter 15

File Documentation

15.1 /home/vm/mark3/trunk/embedded/kernel/atomic.cpp File Reference

Basic Atomic Operations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "atomic.h"
#include "threadport.h"
#include "dbg_file_list.h"
#include "buffalogger.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 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]-------------------------------------
00028 #include "dbg_file_list.h"
00029 #include "buffalogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined!  Bailing."
00032 #else
00033 # define DBG_FILE _DBG___KERNEL_ATOMIC_CPP
00034 #endif
00035 //--[End Autogenerated content]-----------------------------------------
00036
00037 #if KERNEL_USE_ATOMIC
```

```
00038
00039 //---------------------------------------------------------------------------
00040 uint8_t Atomic::Set( uint8_t *pu8Source_, uint8_t u8Val_ )
00041 {
00042     uint8_t u8Ret;
00043     CS_ENTER();
00044     u8Ret = *pu8Source_;
00045     *pu8Source_ = u8Val_;
00046     CS_EXIT();
00047     return u8Ret;
00048 }
00049 //---------------------------------------------------------------------------
00050 uint16_t Atomic::Set( uint16_t *pu16Source_, uint16_t u16Val_ )
00051 {
00052     uint16_t u16Ret;
00053     CS_ENTER();
00054     u16Ret = *pu16Source_;
00055     *pu16Source_ = u16Val_;
00056     CS_EXIT();
00057     return u16Ret;
00058 }
00059 //---------------------------------------------------------------------------
00060 uint32_t Atomic::Set( uint32_t *pu32Source_, uint32_t u32Val_ )
00061 {
00062     uint32_t u32Ret;
00063     CS_ENTER();
00064     u32Ret = *pu32Source_;
00065     *pu32Source_ = u32Val_;
00066     CS_EXIT();
00067     return u32Ret;
00068 }
00069
00070 //---------------------------------------------------------------------------
00071 uint8_t Atomic::Add( uint8_t *pu8Source_, uint8_t u8Val_ )
00072 {
00073     uint8_t u8Ret;
00074     CS_ENTER();
00075     u8Ret = *pu8Source_;
00076     *pu8Source_ += u8Val_;
00077     CS_EXIT();
00078     return u8Ret;
00079 }
00080
00081 //---------------------------------------------------------------------------
00082 uint16_t Atomic::Add( uint16_t *pu16Source_, uint16_t u16Val_ )
00083 {
00084     uint16_t u16Ret;
00085     CS_ENTER();
00086     u16Ret = *pu16Source_;
00087     *pu16Source_ += u16Val_;
00088     CS_EXIT();
00089     return u16Ret;
00090 }
00091
00092 //---------------------------------------------------------------------------
00093 uint32_t Atomic::Add( uint32_t *pu32Source_, uint32_t u32Val_ )
00094 {
00095     uint32_t u32Ret;
00096     CS_ENTER();
00097     u32Ret = *pu32Source_;
00098     *pu32Source_ += u32Val_;
00099     CS_EXIT();
00100     return u32Ret;
00101 }
00102
00103 //---------------------------------------------------------------------------
00104 uint8_t Atomic::Sub( uint8_t *pu8Source_, uint8_t u8Val_ )
00105 {
00106     uint8_t u8Ret;
00107     CS_ENTER();
00108     u8Ret = *pu8Source_;
00109     *pu8Source_ -= u8Val_;
00110     CS_EXIT();
00111     return u8Ret;
00112 }
00113
00114 //---------------------------------------------------------------------------
00115 uint16_t Atomic::Sub( uint16_t *pu16Source_, uint16_t u16Val_ )
00116 {
00117     uint16_t u16Ret;
00118     CS_ENTER();
00119     u16Ret = *pu16Source_;
00120     *pu16Source_ -= u16Val_;
00121     CS_EXIT();
00122     return u16Ret;
00123 }
00124
```

```
00125 //---------------------------------------------------------------------------
00126 uint32_t Atomic::Sub( uint32_t *pu32Source_, uint32_t u32Val_ )
00127 {
00128     uint32_t u32Ret;
00129     CS_ENTER();
00130     u32Ret = *pu32Source_;
00131     *pu32Source_ -= u32Val_;
00132     CS_EXIT();
00133     return u32Ret;
00134 }
00135
00136 //---------------------------------------------------------------------------
00137 bool Atomic::TestAndSet( bool *pbLock_ )
00138 {
00139     uint8_t u8Ret;
00140     CS_ENTER();
00141     u8Ret = *pbLock_;
00142     if (!u8Ret)
00143     {
00144         *pbLock_ = 1;
00145     }
00146     CS_EXIT();
00147     return u8Ret;
00148 }
00149
00150 #endif // KERNEL_USE_ATOMIC
```

15.3 /home/vm/mark3/trunk/embedded/kernel/blocking.cpp File Reference

Implementation of base class for blocking objects.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
#include "thread.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

15.3.1 Detailed Description

Implementation of base class for blocking objects.

Definition in file blocking.cpp.

15.4 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
00024 #include "blocking.h"
00025 #include "thread.h"
00026
00027 #define _CAN_HAS_DEBUG
00028 //--[Autogenerated - Do Not Modify]-------------------------------------
00029 #include "dbg_file_list.h"
00030 #include "buffalogger.h"
```

```
00031 #if defined(DBG_FILE)
00032 # error "Debug logging file token already defined!  Bailing."
00033 #else
00034 # define DBG_FILE _DBG___KERNEL_BLOCKING_CPP
00035 #endif
00036 //--[End Autogenerated content]-----------------------------------------------
00037 #include "kerneldebug.h"
00038
00039 #if KERNEL_USE_SEMAPHORE || KERNEL_USE_MUTEX
00040 //---------------------------------------------------------------------------
00041 void BlockingObject::Block(Thread *pclThread_)
00042 {
00043     KERNEL_ASSERT( pclThread_ );
00044     KERNEL_TRACE_1( "Blocking Thread %d", (uint16_t)pclThread_->GetID() );
00045
00046     // Remove the thread from its current thread list (the "owner" list)
00047     // ... And add the thread to this object's block list
00048     Scheduler::Remove(pclThread_);
00049     m_clBlockList.Add(pclThread_);
00050
00051     // Set the "current" list location to the blocklist for this thread
00052     pclThread_->SetCurrent(&m_clBlockList);
00053     pclThread_->SetState(THREAD_STATE_BLOCKED);
00054 }
00055
00056 //---------------------------------------------------------------------------
00057 void BlockingObject::UnBlock(Thread *pclThread_)
00058 {
00059     KERNEL_ASSERT( pclThread_ );
00060     KERNEL_TRACE_1( "Unblocking Thread %d", (uint16_t)pclThread_->GetID() );
00061
00062     // Remove the thread from its current thread list (the "owner" list)
00063     pclThread_->GetCurrent()->Remove(pclThread_);
00064
00065     // Put the thread back in its active owner's list.  This is usually
00066     // the ready-queue at the thread's original priority.
00067     Scheduler::Add(pclThread_);
00068
00069     // Tag the thread's current list location to its owner
00070     pclThread_->SetCurrent(pclThread_->GetOwner());
00071     pclThread_->SetState(THREAD_STATE_READY);
00072 }
00073
00074 #endif
```

15.5 /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp File Reference

ATMega328p Profiling timer implementation.

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

15.5.1 Detailed Description

ATMega328p Profiling timer implementation.

Definition in file kernelprofile.cpp.

15.6 kernelprofile.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 "kernelprofile.h"
00024 #include "threadport.h"
00025 #include <avr/io.h>
00026 #include <avr/interrupt.h>
00027
00028 #if KERNEL_USE_PROFILER
00029 uint32_t Profiler::m_u32Epoch;
00030
00031 //---------------------------------------------------------------------------
00032 void Profiler::Init()
00033 {
00034     TCCR0A = 0;
00035     TCCR0B = 0;
00036     TIFR0 = 0;
00037     TIMSK0 = 0;
00038     m_u32Epoch = 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 uint16_t Profiler::Read()
00059 {
00060     uint16_t u16Ret;
00061     CS_ENTER();
00062     TCCR0B &= ~(1 << CS01);
00063     u16Ret = TCNT0;
00064     TCCR0B |= (1 << CS01);
00065     CS_EXIT();
00066     return u16Ret;
00067 }
00068
00069 //---------------------------------------------------------------------------
00070 void Profiler::Process()
00071 {
00072     CS_ENTER();
00073     m_u32Epoch++;
00074     CS_EXIT();
00075 }
00076
00077 //---------------------------------------------------------------------------
00078 ISR(TIMER0_OVF_vect)
00079 {
00080     Profiler::Process();
00081 }
00082
00083 #endif
```

15.7 /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp File Reference

Kernel Software interrupt implementation for ATMega328p.

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

15.7.1 Detailed Description

Kernel Software interrupt implementation for ATMega328p.

Definition in file kernelswi.cpp.

15.8 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 int32_t 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 uint8_t 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 }
```

15.9 /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/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>
```

15.9.1 Detailed Description

Kernel Timer Implementation for ATMega328p.

Definition in file kerneltimer.cpp.

15.10 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 uint16_t KernelTimer::Read(void)
00067 {
00068 #if KERNEL_TIMERS_TICKLESS
00069     volatile uint16_t u16Read1;
00070     volatile uint16_t u16Read2;
00071
00072     do {
00073         u16Read1 = TCNT1;
00074         u16Read2 = TCNT1;
00075     } while (u16Read1 != u16Read2);
00076
00077     return u16Read1;
00078 #else
00079     return 0;
00080 #endif
00081 }
00082
00083 //---------------------------------------------------------------------------
00084 uint32_t KernelTimer::SubtractExpiry(uint32_t u32Interval_)
00085 {
00086 #if KERNEL_TIMERS_TICKLESS
00087     OCR1A -= (uint16_t)u32Interval_;
00088     return (uint32_t)OCR1A;
00089 #else
00090     return 0;
00091 #endif
00092 }
00093
00094 //---------------------------------------------------------------------------
00095 uint32_t KernelTimer::TimeToExpiry(void)
00096 {
00097 #if KERNEL_TIMERS_TICKLESS
00098     uint16_t u16Read = KernelTimer::Read();
00099     uint16_t u16OCR1A = OCR1A;
00100
00101     if (u16Read >= u16OCR1A)
00102     {
00103         return 0;
00104     }
00105     else
00106     {
00107         return (uint32_t)(u16OCR1A - u16Read);
00108     }
00109 #else
00110     return 0;
00111 #endif
00112 }
00113
00114 //---------------------------------------------------------------------------
00115 uint32_t KernelTimer::GetOvertime(void)
00116 {
00117     return KernelTimer::Read();
00118 }
00119
00120 //---------------------------------------------------------------------------
00121 uint32_t KernelTimer::SetExpiry(uint32_t u32Interval_)
00122 {
00123 #if KERNEL_TIMERS_TICKLESS
00124     uint16_t u16SetInterval;
00125     if (u32Interval_ > 65535)
00126     {
00127         u16SetInterval = 65535;
00128     }
00129     else
00130     {
00131         u16SetInterval = (uint16_t)u32Interval_ ;
```

```
00132     }
00133
00134     OCR1A = u16SetInterval;
00135     return (uint32_t)u16SetInterval;
00136 #else
00137     return 0;
00138 #endif
00139 }
00140
00141 //---------------------------------------------------------------------------
00142 void KernelTimer::ClearExpiry(void)
00143 {
00144 #if KERNEL_TIMERS_TICKLESS
00145     OCR1A = 65535;                    // Clear the compare value
00146 #endif
00147 }
00148
00149 //---------------------------------------------------------------------------
00150 uint8_t KernelTimer::DI(void)
00151 {
00152 #if KERNEL_TIMERS_TICKLESS
00153     bool bEnabled = ((TIMSK1 & (TIMER_IMSK)) != 0);
00154     TIFR1 &= ~TIMER_IFR;      // Clear interrupt flags
00155     TIMSK1 &= ~TIMER_IMSK;    // Disable interrupt
00156     return bEnabled;
00157 #else
00158     return 0;
00159 #endif
00160 }
00161
00162 //---------------------------------------------------------------------------
00163 void KernelTimer::EI(void)
00164 {
00165     KernelTimer::RI(0);
00166 }
00167
00168 //---------------------------------------------------------------------------
00169 void KernelTimer::RI(bool bEnable_)
00170 {
00171 #if KERNEL_TIMERS_TICKLESS
00172     if (bEnable_)
00173     {
00174         TIMSK1 |= (1 << OCIE1A);    // Enable interrupt
00175     }
00176     else
00177     {
00178         TIMSK1 &= ~(1 << OCIE1A);
00179     }
00180 #endif
00181 }
```

15.11 /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h File Reference

Profiling timer hardware interface.

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

Classes

- class Profiler

  System profiling timer interface.

15.11.1 Detailed Description

Profiling timer hardware interface.

Definition in file kernelprofile.h.

15.12 kernelprofile.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 uint16_t Read();
00068
00072     static void Process();
00073
00077     static uint32_t GetEpoch(){ return m_u32Epoch; }
00078 private:
00079
00080     static uint32_t m_u32Epoch;
00081 };
00082
00083 #endif //KERNEL_USE_PROFILER
00084
00085 #endif
00086
```

15.13 /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/public/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.

15.13.1 Detailed Description

Kernel Software interrupt declarations.

Definition in file kernelswi.h.

15.14 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 uint8_t DI();
00079
00087     static void RI(bool bEnable_);
00088 };
00089
00090
00091 #endif // __KERNELSIW_H_
```

15.15 /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h File Reference

Kernel Timer Class declaration.

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

Classes

- class KernelTimer

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

15.15.1 Detailed Description

Kernel Timer Class declaration.

Definition in file kerneltimer.h.

15.16 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        ((uint32_t)16000000)
00027 #define TIMER_FREQ        ((uint32_t)(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 uint8_t DI(void);
00063
00071     static void RI(bool bEnable_);
00072
00078     static void EI(void);
00079
00090     static uint32_t SubtractExpiry(uint32_t u32Interval_);
00091
00100     static uint32_t TimeToExpiry(void);
00101
00110     static uint32_t SetExpiry(uint32_t u32Interval_);
00111
00120     static uint32_t GetOvertime(void);
00121
00127     static void ClearExpiry(void);
00128
00129 private:
00137     static uint16_t Read(void);
00138
00139 };
00140
00141 #endif //__KERNELTIMER_H_
```

15.17 /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/public/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 TOP_OF_STACK(x, y) (uint8_t∗) (((uint16_t)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.

15.17.1 Detailed Description

ATMega328p Multithreading support.

Definition in file threadport.h.

15.17.2 Macro Definition Documentation

15.17.2.1 #define CS_ENTER( )

Value:

```
{ \
volatile uint8_t 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.

15.18 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)        (uint8_t*) ( ((uint16_t)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_pclCurrent"); \
00086 ASM("lds r27, g_pclCurrent + 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_pclCurrent"); \
00097 ASM("lds r27, g_pclCurrent + 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 uint8_t 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_
```

15.19 /home/vm/mark3/trunk/embedded/kernel/cpu/avr/atmega328p/gcc/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 "kernel.h"
#include "kernelaware.h"
#include <avr/io.h>
#include <avr/interrupt.h>
```

Functions

- 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.

15.19.1 Detailed Description

ATMega328p Multithreading.

Definition in file threadport.cpp.

15.20 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 "kernel.h"
00031 #include "kernelaware.h"
00032 #include <avr/io.h>
00033 #include <avr/interrupt.h>
00034
00035 //---------------------------------------------------------------------------
00036 Thread *g_pclCurrentThread;
00037
00038 //---------------------------------------------------------------------------
00039 void ThreadPort::InitStack(Thread *pclThread_)
00040 {
00041     // Initialize the stack for a Thread
00042     uint16_t u16Addr;
00043     uint8_t *pu8Stack;
00044     uint16_t i;
00045
00046     // Get the address of the thread's entry function
00047     u16Addr = (uint16_t)(pclThread_->m_pfEntryPoint);
00048
00049     // Start by finding the bottom of the stack
00050     pu8Stack = (uint8_t*)pclThread_->m_pwStackTop;
00051
00052     // clear the stack, and initialize it to a known-default value (easier
00053     // to debug when things go sour with stack corruption or overflow)
00054     for (i = 0; i < pclThread_->m_u16StackSize; i++)
00055     {
00056         pclThread_->m_pwStack[i] = 0xFF;
00057     }
00058
00059     // Our context starts with the entry function
00060     PUSH_TO_STACK(pu8Stack, (uint8_t)(u16Addr & 0x00FF));
00061     PUSH_TO_STACK(pu8Stack, (uint8_t)((u16Addr >> 8) & 0x00FF));
00062
00063     // R0
00064     PUSH_TO_STACK(pu8Stack, 0x00);    // R0
00065
00066     // Push status register and R1 (which is used as a constant zero)
00067     PUSH_TO_STACK(pu8Stack, 0x80);  // SR
00068     PUSH_TO_STACK(pu8Stack, 0x00);  // R1
00069
00070     // Push other registers
00071     for (i = 2; i <= 23; i++) //R2-R23
00072     {
00073         PUSH_TO_STACK(pu8Stack, i);
00074     }
00075
00076     // Assume that the argument is the only stack variable
00077     PUSH_TO_STACK(pu8Stack, (uint8_t)(((uint16_t)(pclThread_->
```

```
m_pvArg)) & 0x00FF));    //R24
00078     PUSH_TO_STACK(pu8Stack, (uint8_t)((((uint16_t)(pclThread_->
m_pvArg))>>8) & 0x00FF)); //R25
00079
00080     // Push the rest of the registers in the context
00081     for (i = 26; i <=31; i++)
00082     {
00083         PUSH_TO_STACK(pu8Stack, i);
00084     }
00085
00086     // Set the top o' the stack.
00087     pclThread_->m_pwStackTop = (uint8_t*)pu8Stack;
00088
00089     // That's it!  the thread is ready to run now.
00090 }
00091
00092 //---------------------------------------------------------------------------
00093 static void Thread_Switch(void)
00094 {
00095 #if KERNEL_USE_IDLE_FUNC
00096     // If there's no next-thread-to-run...
00097     if (g_pclNext == Kernel::GetIdleThread())
00098     {
00099         g_pclCurrent = Kernel::GetIdleThread();
00100
00101         // Disable the SWI, and re-enable interrupts -- enter nested interrupt
00102         // mode.
00103         KernelSWI::DI();
00104
00105         uint8_t u8SR = _SFR_IO8(SR_);
00106
00107         // So long as there's no "next-to-run" thread, keep executing the Idle
00108         // function to conclusion...
00109
00110         while (g_pclNext == Kernel::GetIdleThread())
00111         {
00112            // Ensure that we run this block in an interrupt enabled context (but
00113            // with the rest of the checks being performed in an interrupt disabled
00114            // context).
00115            ASM( "sei" );
00116            Kernel::IdleFunc();
00117            ASM( "cli" );
00118         }
00119
00120         // Progress has been achieved -- an interrupt-triggered event has caused
00121         // the scheduler to run, and choose a new thread.  Since we've already
00122         // saved the context of the thread we've hijacked to run idle, we can
00123         // proceed to disable the nested interrupt context and switch to the
00124         // new thread.
00125
00126         _SFR_IO8(SR_) = u8SR;
00127         KernelSWI::RI( true );
00128     }
00129 #endif
00130     g_pclCurrent = (Thread*)g_pclNext;
00131 }
00132
00133
00134 //---------------------------------------------------------------------------
00135 void ThreadPort::StartThreads()
00136 {
00137     KernelSWI::Config();                // configure the task switch SWI
00138     KernelTimer::Config();              // configure the kernel timer
00139
00140     Scheduler::SetScheduler(1);         // enable the scheduler
00141     Scheduler::Schedule();              // run the scheduler - determine the first
     thread to run
00142
00143     Thread_Switch();                    // Set the next scheduled thread to the current thread
00144
00145     KernelTimer::Start();               // enable the kernel timer
00146     KernelSWI::Start();                 // enable the task switch SWI
00147
00148     // Restore the context...
00149     Thread_RestoreContext();        // restore the context of the first running thread
00150     ASM("reti");                    // return from interrupt - will return to the first scheduled thread
00151 }
00152
00153 //---------------------------------------------------------------------------
00158 //---------------------------------------------------------------------------
00159 ISR(INT0_vect) __attribute__ ( ( signal, naked ) );
00160 ISR(INT0_vect)
00161 {
00162     Thread_SaveContext();       // Push the context (registers) of the current task
00163     Thread_Switch();            // Switch to the next task
00164     Thread_RestoreContext();    // Pop the context (registers) of the next task
00165     ASM("reti");                // Return to the next task
```

```
00166 }
00167
00168 //---------------------------------------------------------------------------
00173 //---------------------------------------------------------------------------
00174 ISR(TIMER1_COMPA_vect)
00175 {
00176 #if KERNEL_USE_TIMERS
00177     TimerScheduler::Process();
00178 #endif
00179 #if KERNEL_USE_QUANTUM
00180     Quantum::UpdateTimer();
00181 #endif
00182 }
```

15.21 /home/vm/mark3/trunk/embedded/kernel/driver.cpp File Reference

Device driver/hardware abstraction layer.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "driver.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

Classes

- class DevNull

  This class implements the "default" driver (/dev/null)

Functions

- static uint8_t DrvCmp (const char *szStr1_, const char *szStr2_)

  DrvCmp.

Variables

- static DevNull clDevNull

  Default driver included to allow for run-time "stubbing".

15.21.1 Detailed Description

Device driver/hardware abstraction layer.

Definition in file driver.cpp.

15.21.2 Function Documentation

15.21.2.1 static uint8_t DrvCmp (const char * *szStr1_*, const char * *szStr2_*) [static]

DrvCmp.

String comparison function used to compare input driver name against a known driver name in the existing driver list.

Parameters

*szStr1_*	user-specified driver name
*szStr2_*	name of a driver, provided from the driver table

Returns

1 on match, 0 on no-match

Definition at line 81 of file driver.cpp.

15.22 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 "driver.h"
00024
00025 #define _CAN_HAS_DEBUG
00026 //--[Autogenerated - Do Not Modify]-------------------------------------
00027 #include "dbg_file_list.h"
00028 #include "buffalogger.h"
00029 #if defined(DBG_FILE)
00030 # error "Debug logging file token already defined!  Bailing."
00031 #else
00032 # define DBG_FILE _DBG___KERNEL_DRIVER_CPP
00033 #endif
00034 //--[End Autogenerated content]-----------------------------------------
00035
00036 #include "kerneldebug.h"
00037
00038 //---------------------------------------------------------------------------
00039 #if KERNEL_USE_DRIVER
00040
00041 DoubleLinkList DriverList::m_clDriverList;
00042
00046 class DevNull : public Driver
00047 {
00048 public:
00049     virtual void Init() { SetName("/dev/null"); };
00050     virtual uint8_t Open() { return 0; }
00051     virtual uint8_t Close() { return 0; }
00052
00053     virtual uint16_t Read( uint16_t u16Bytes_,
00054     uint8_t *pu8Data_){ return 0; }
00055
00056     virtual uint16_t Write( uint16_t u16Bytes_,
00057     uint8_t *pu8Data_) { return 0; }
00058
00059     virtual uint16_t Control( uint16_t u16Event_,
00060         void *pvDataIn_,
00061         uint16_t u16SizeIn_,
00062         void *pvDataOut_,
00063         uint16_t u16SizeOut_ ) { return 0; }
00064
00065 };
00066
00067 //---------------------------------------------------------------------------
00068 static DevNull clDevNull;
00069
00070 //---------------------------------------------------------------------------
00081 static uint8_t DrvCmp( const char *szStr1_, const char *szStr2_ )
00082 {
00083     char *szTmp1 = (char*) szStr1_;
00084     char *szTmp2 = (char*) szStr2_;
00085
00086     while (*szTmp1 && *szTmp2)
```

```
00087     {
00088         if (*szTmp1++ != *szTmp2++)
00089         {
00090             return 0;
00091         }
00092     }
00093 
00094     // Both terminate at the same length
00095     if (!(*szTmp1) && !(*szTmp2))
00096     {
00097         return 1;
00098     }
00099 
00100     return 0;
00101 }
00102 
00103 //---------------------------------------------------------------------------
00104 void DriverList::Init()
00105 {
00106     // Ensure we always have at least one entry - a default in case no match
00107     // is found (/dev/null)
00108     clDevNull.Init();
00109     Add(&clDevNull);
00110 }
00111 
00112 //---------------------------------------------------------------------------
00113 Driver *DriverList::FindByPath( const char *m_pcPath )
00114 {
00115     KERNEL_ASSERT( m_pcPath );
00116     Driver *pclTemp = static_cast<Driver*>(m_clDriverList.
      GetHead());
00117 
00118     // Iterate through the list of drivers until we find a match, or we
00119     // exhaust our list of installed drivers
00120     while (pclTemp)
00121     {
00122         if(DrvCmp(m_pcPath, pclTemp->GetPath()))
00123         {
00124             return pclTemp;
00125         }
00126         pclTemp = static_cast<Driver*>(pclTemp->GetNext());
00127     }
00128     // No matching driver found - return a pointer to our /dev/null driver
00129     return &clDevNull;
00130 }
00131 
00132 #endif
```

15.23 /home/vm/mark3/trunk/embedded/kernel/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"
#include "kernelaware.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "timerlist.h"
```

Functions

- void TimedEventFlag_Callback (Thread *pclOwner_, void *pvData_)

  TimedEventFlag_Callback.

15.23.1 Detailed Description

Event Flag Blocking Object/IPC-Object implementation.

Definition in file eventflag.cpp.

15.23.2 Function Documentation

15.23.2.1 void TimedEventFlag_Callback (Thread * *pclOwner_*, void * *pvData_*)

TimedEventFlag_Callback.

This funciton is called whenever a timed event flag wait operation fails in the time provided. This function wakes the thread for which the timeout was requested on the blocking call, sets the thread's expiry flags, and reschedules if necessary.

Parameters

*pclOwner_*	Thread to wake
*pvData_*	Pointer to the event-flag object

Definition at line 53 of file eventflag.cpp.

15.24 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 #include "kernelaware.h"
00025
00026 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]-------------------------------------
00028 #include "dbg_file_list.h"
00029 #include "buffalogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined!  Bailing."
00032 #else
00033 # define DBG_FILE _DBG___KERNEL_EVENTFLAG_CPP
00034 #endif
00035 //--[End Autogenerated content]-----------------------------------------
00036
00037 #if KERNEL_USE_EVENTFLAG
00038
00039 #if KERNEL_USE_TIMEOUTS
00040 #include "timerlist.h"
00041 //---------------------------------------------------------------------------
00053 void TimedEventFlag_Callback(Thread *pclOwner_, void *pvData_)
00054 {
00055     EventFlag *pclEventFlag = static_cast<EventFlag*>(pvData_);
00056
00057     pclEventFlag->WakeMe(pclOwner_);
00058     pclOwner_->SetExpired(true);
00059     pclOwner_->SetEventFlagMask(0);
00060
00061     if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
      ()->GetCurPriority())
00062     {
00063         Thread::Yield();
00064     }
00065 }
00066
00067 //---------------------------------------------------------------------------
00068 void EventFlag::WakeMe(Thread *pclChosenOne_)
00069 {
00070     UnBlock(pclChosenOne_);
```

```
00071 }
00072 #endif
00073
00074 //---------------------------------------------------------------------------
00075 #if KERNEL_USE_TIMEOUTS
00076     uint16_t EventFlag::Wait_i(uint16_t u16Mask_,
      EventFlagOperation_t eMode_, uint32_t u32TimeMS_)
00077 #else
00078     uint16_t EventFlag::Wait_i(uint16_t u16Mask_,
      EventFlagOperation_t eMode_)
00079 #endif
00080 {
00081     bool bThreadYield = false;
00082     bool bMatch = false;
00083
00084 #if KERNEL_USE_TIMEOUTS
00085     Timer clEventTimer;
00086     bool bUseTimer = false;
00087 #endif
00088
00089     // Ensure we're operating in a critical section while we determine
00090     // whether or not we need to block the current thread on this object.
00091     CS_ENTER();
00092
00093     // Check to see whether or not the current mask matches any of the
00094     // desired bits.
00095     g_pclCurrent->SetEventFlagMask(u16Mask_);
00096
00097     if ((eMode_ == EVENT_FLAG_ALL) || (eMode_ ==
      EVENT_FLAG_ALL_CLEAR))
00098     {
00099         // Check to see if the flags in their current state match all of
00100         // the set flags in the event flag group, with this mask.
00101         if ((m_u16SetMask & u16Mask_) == u16Mask_)
00102         {
00103             bMatch = true;
00104             g_pclCurrent->SetEventFlagMask(u16Mask_);
00105         }
00106     }
00107     else if ((eMode_ == EVENT_FLAG_ANY) || (eMode_ ==
      EVENT_FLAG_ANY_CLEAR))
00108     {
00109         // Check to see if the existing flags match any of the set flags in
00110         // the event flag group  with this mask
00111         if (m_u16SetMask & u16Mask_)
00112         {
00113             bMatch = true;
00114             g_pclCurrent->SetEventFlagMask(m_u16SetMask & u16Mask_);
00115         }
00116     }
00117
00118     // We're unable to match this pattern as-is, so we must block.
00119     if (!bMatch)
00120     {
00121         // Reset the current thread's event flag mask & mode
00122         g_pclCurrent->SetEventFlagMask(u16Mask_);
00123         g_pclCurrent->SetEventFlagMode(eMode_);
00124
00125 #if KERNEL_USE_TIMEOUTS
00126         if (u32TimeMS_)
00127         {
00128             g_pclCurrent->SetExpired(false);
00129             clEventTimer.Init();
00130             clEventTimer.Start(0, u32TimeMS_, TimedEventFlag_Callback, (void*)
      this);
00131             bUseTimer = true;
00132         }
00133 #endif
00134
00135         // Add the thread to the object's block-list.
00136         Block(g_pclCurrent);
00137
00138         // Trigger that
00139         bThreadYield = true;
00140     }
00141
00142     // If bThreadYield is set, it means that we've blocked the current thread,
00143     // and must therefore rerun the scheduler to determine what thread to
00144     // switch to.
00145     if (bThreadYield)
00146     {
00147         // Switch threads immediately
00148         Thread::Yield();
00149     }
00150
00151     // Exit the critical section and return back to normal execution
00152     CS_EXIT();
```

```
00153
00158 #if KERNEL_USE_TIMEOUTS
00159     if (bUseTimer && bThreadYield)
00160     {
00161         clEventTimer.Stop();
00162     }
00163 #endif
00164
00165     return g_pclCurrent->GetEventFlagMask();
00166 }
00167
00168 //---------------------------------------------------------------------------
00169 uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_)
00170 {
00171 #if KERNEL_USE_TIMEOUTS
00172     return Wait_i(u16Mask_, eMode_, 0);
00173 #else
00174     return Wait_i(u16Mask_, eMode_);
00175 #endif
00176 }
00177
00178 #if KERNEL_USE_TIMEOUTS
00179 //---------------------------------------------------------------------------
00180 uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_,
      uint32_t u32TimeMS_)
00181 {
00182     return Wait_i(u16Mask_, eMode_, u32TimeMS_);
00183 }
00184 #endif
00185
00186 //---------------------------------------------------------------------------
00187 void EventFlag::Set(uint16_t u16Mask_)
00188 {
00189     Thread *pclPrev;
00190     Thread *pclCurrent;
00191     bool bReschedule = false;
00192     uint16_t u16NewMask;
00193
00194     CS_ENTER();
00195
00196     // Walk through the whole block list, checking to see whether or not
00197     // the current flag set now matches any/all of the masks and modes of
00198     // the threads involved.
00199
00200     m_u16SetMask |= u16Mask_;
00201     u16NewMask = m_u16SetMask;
00202
00203     // Start at the head of the list, and iterate through until we hit the
00204     // "head" element in the list again.  Ensure that we handle the case where
00205     // we remove the first or last elements in the list, or if there's only
00206     // one element in the list.
00207     pclCurrent = static_cast<Thread*>(m_clBlockList.GetHead());
00208
00209     // Do nothing when there are no objects blocking.
00210     if (pclCurrent)
00211     {
00212         // First loop - process every thread in the block-list and check to
00213         // see whether or not the current flags match the event-flag conditions
00214         // on the thread.
00215         do
00216         {
00217             pclPrev = pclCurrent;
00218             pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00219
00220             // Read the thread's event mask/mode
00221             uint16_t u16ThreadMask = pclPrev->GetEventFlagMask();
00222             EventFlagOperation_t eThreadMode = pclPrev->
      GetEventFlagMode();
00223
00224             // For the "any" mode - unblock the blocked threads if one or more bits
00225             // in the thread's bitmask match the object's bitmask
00226             if ((EVENT_FLAG_ANY == eThreadMode) || (
      EVENT_FLAG_ANY_CLEAR == eThreadMode))
00227             {
00228                 if (u16ThreadMask & m_u16SetMask)
00229                 {
00230                     pclPrev->SetEventFlagMode(
      EVENT_FLAG_PENDING_UNBLOCK);
00231                     pclPrev->SetEventFlagMask(m_u16SetMask & u16ThreadMask);
00232                     bReschedule = true;
00233
00234                     // If the "clear" variant is set, then clear the bits in the mask
00235                     // that caused the thread to unblock.
00236                     if (EVENT_FLAG_ANY_CLEAR == eThreadMode)
00237                     {
00238                         u16NewMask &=~ (u16ThreadMask & u16Mask_);
00239                     }
```

```
00240                 }
00241             }
00242             // For the "all" mode, every set bit in the thread's requested bitmask must
00243             // match the object's flag mask.
00244             else if ((EVENT_FLAG_ALL == eThreadMode) || (
      EVENT_FLAG_ALL_CLEAR == eThreadMode))
00245             {
00246                 if ((u16ThreadMask & m_u16SetMask) == u16ThreadMask)
00247                 {
00248                     pclPrev->SetEventFlagMode(
      EVENT_FLAG_PENDING_UNBLOCK);
00249                     pclPrev->SetEventFlagMask(u16ThreadMask);
00250                     bReschedule = true;
00251 
00252                     // If the "clear" variant is set, then clear the bits in the mask
00253                     // that caused the thread to unblock.
00254                     if (EVENT_FLAG_ALL_CLEAR == eThreadMode)
00255                     {
00256                         u16NewMask &=~ (u16ThreadMask & u16Mask_);
00257                     }
00258                 }
00259             }
00260         }
00261         // To keep looping, ensure that there's something in the list, and
00262         // that the next item isn't the head of the list.
00263         while (pclPrev != m_clBlockList.GetTail());
00264 
00265         // Second loop - go through and unblock all of the threads that
00266         // were tagged for unblocking.
00267         pclCurrent = static_cast<Thread*>(m_clBlockList.
      GetHead());
00268         bool bIsTail = false;
00269         do
00270         {
00271             pclPrev = pclCurrent;
00272             pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00273 
00274             // Check to see if this is the condition to terminate the loop
00275             if (pclPrev == m_clBlockList.GetTail())
00276             {
00277                 bIsTail = true;
00278             }
00279 
00280             // If the first pass indicated that this thread should be
00281             // unblocked, then unblock the thread
00282             if (pclPrev->GetEventFlagMode() ==
      EVENT_FLAG_PENDING_UNBLOCK)
00283             {
00284                 UnBlock(pclPrev);
00285             }
00286         }
00287         while (!bIsTail);
00288     }
00289 
00290     // If we awoke any threads, re-run the scheduler
00291     if (bReschedule)
00292     {
00293         Thread::Yield();
00294     }
00295 
00296     // Update the bitmask based on any "clear" operations performed along
00297     // the way
00298     m_u16SetMask = u16NewMask;
00299 
00300     // Restore interrupts - will potentially cause a context switch if a
00301     // thread is unblocked.
00302     CS_EXIT();
00303 }
00304 
00305 //---------------------------------------------------------------------------
00306 void EventFlag::Clear(uint16_t u16Mask_)
00307 {
00308     // Just clear the bitfields in the local object.
00309     CS_ENTER();
00310     m_u16SetMask &= ~u16Mask_;
00311     CS_EXIT();
00312 }
00313 
00314 //---------------------------------------------------------------------------
00315 uint16_t EventFlag::GetMask()
00316 {
00317     // Return the presently held event flag values in this object.  Ensure
00318     // we get this within a critical section to guarantee atomicity.
00319     uint16_t u16Return;
00320     CS_ENTER();
00321     u16Return = m_u16SetMask;
00322     CS_EXIT();
```

```
00323     return u16Return;
00324 }
00325
00326 #endif // KERNEL_USE_EVENTFLAG
```

15.25 /home/vm/mark3/trunk/embedded/kernel/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 "kernelprofile.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
#include "tracebuffer.h"
```

15.25.1 Detailed Description

Kernel initialization and startup code.

Definition in file kernel.cpp.

15.26 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 "kernelprofile.h"
00033
00034
00035 #define _CAN_HAS_DEBUG
00036 //--[Autogenerated - Do Not Modify]-----------------------------------------
00037 #include "dbg_file_list.h"
00038 #include "buffalogger.h"
00039 #if defined(DBG_FILE)
00040 # error "Debug logging file token already defined!  Bailing."
```

```
00041 #else
00042 # define DBG_FILE _DBG___KERNEL_KERNEL_CPP
00043 #endif
00044 //--[End Autogenerated content]----------------------------------------------
00045 #include "kerneldebug.h"
00046 #include "tracebuffer.h"
00047
00048 bool Kernel::m_bIsStarted;
00049 bool Kernel::m_bIsPanic;
00050 panic_func_t Kernel::m_pfPanic;
00051
00052 #if KERNEL_USE_IDLE_FUNC
00053 idle_func_t Kernel::m_pfIdle;
00054 FakeThread_t Kernel::m_clIdle;
00055 #endif
00056 //---------------------------------------------------------------------------
00057 void Kernel::Init(void)
00058 {
00059     m_bIsStarted = false;
00060     m_bIsPanic = false;
00061     m_pfPanic = 0;
00062
00063 #if KERNEL_USE_IDLE_FUNC
00064     ((Thread*)&m_clIdle)->InitIdle();
00065     m_pfIdle = 0;
00066 #endif
00067
00068 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00069     TraceBuffer::Init();
00070 #endif
00071     KERNEL_TRACE( "Initializing Mark3 Kernel" );
00072
00073     // Initialize the global kernel data - scheduler, timer-scheduler, and
00074     // the global message pool.
00075     Scheduler::Init();
00076 #if KERNEL_USE_DRIVER
00077     DriverList::Init();
00078 #endif
00079 #if KERNEL_USE_TIMERS
00080     TimerScheduler::Init();
00081 #endif
00082 #if KERNEL_USE_MESSAGE
00083     GlobalMessagePool::Init();
00084 #endif
00085 #if KERNEL_USE_PROFILER
00086     Profiler::Init();
00087 #endif
00088 }
00089
00090 //---------------------------------------------------------------------------
00091 void Kernel::Start(void)
00092 {
00093     KERNEL_TRACE( "Starting Mark3 Scheduler" );
00094     m_bIsStarted = true;
00095     ThreadPort::StartThreads();
00096     KERNEL_TRACE( "Error starting Mark3 Scheduler" );
00097 }
00098
00099 //---------------------------------------------------------------------------
00100 void Kernel::Panic(uint16_t u16Cause_)
00101 {
00102     m_bIsPanic = true;
00103     if (m_pfPanic)
00104     {
00105         m_pfPanic(u16Cause_);
00106     }
00107     else
00108     {
00109 #if KERNEL_AWARE_SIMULATION
00110         KernelAware::ExitSimulator();
00111 #endif
00112         while(1);
00113     }
00114 }
```

15.27 /home/vm/mark3/trunk/embedded/kernel/kernelaware.cpp File Reference

Kernel aware simulation support.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "kernelaware.h"
#include "threadport.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
```

Classes

- union KernelAwareData_t

  This structure is used to communicate between the kernel and a kernel- aware host.

Variables

- volatile bool g_bIsKernelAware = false

  Will be set to true by a kernel-aware host.
- volatile uint8_t g_u8KACommand

  Kernel-aware simulator command to execute.
- KernelAwareData_t g_stKAData

  Data structure used to communicate with host.

15.27.1 Detailed Description

Kernel aware simulation support.

Definition in file kernelaware.cpp.

15.27.2 Variable Documentation

15.27.2.1 volatile bool g_bIsKernelAware = false

Will be set to true by a kernel-aware host.

Definition at line 81 of file kernelaware.cpp.

15.27.2.2 KernelAwareData_t g_stKAData

Data structure used to communicate with host.

Definition at line 83 of file kernelaware.cpp.

15.28 kernelaware.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 "kernelaware.h"
00024 #include "threadport.h"
00025
00026 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]-------------------------------------
00028 #include "dbg_file_list.h"
00029 #include "buffalogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined!  Bailing."
00032 #else
00033 # define DBG_FILE _DBG___KERNEL_KERNELAWARE_CPP
00034 #endif
00035 //--[End Autogenerated content]-----------------------------------------
00036
00037 #if KERNEL_AWARE_SIMULATION
00038
00039 //---------------------------------------------------------------------------
00048 typedef union
00049 {
00050     volatile uint16_t au16Buffer[5];
00051
00055     struct
00056     {
00057         volatile const char *szName;
00058     } Profiler;
00063     struct
00064     {
00065         volatile uint16_t u16File;
00066         volatile uint16_t u16Line;
00067         volatile uint16_t u16Arg1;
00068         volatile uint16_t u16Arg2;
00069     } Trace;
00074     struct
00075     {
00076         volatile const char *szString;
00077     } Print;
00078 } KernelAwareData_t;
00079
00080 //---------------------------------------------------------------------------
00081 volatile bool           g_bIsKernelAware = false;
00082 volatile uint8_t        g_u8KACommand;
00083 KernelAwareData_t       g_stKAData;
00084
00085
00086 //---------------------------------------------------------------------------
00087 void KernelAware::ProfileInit(const char *szStr_)
00088 {
00089     CS_ENTER();
00090     g_stKAData.Profiler.szName = szStr_;
00091     g_u8KACommand = KA_COMMAND_PROFILE_INIT;
00092     CS_EXIT();
00093 }
00094
00095 //---------------------------------------------------------------------------
00096 void KernelAware::ProfileStart(void)
00097 {
00098     g_u8KACommand = KA_COMMAND_PROFILE_START;
00099 }
00100
00101 //---------------------------------------------------------------------------
00102 void KernelAware::ProfileStop(void)
00103 {
00104     g_u8KACommand = KA_COMMAND_PROFILE_STOP;
00105 }
00106
00107 //---------------------------------------------------------------------------
00108 void KernelAware::ProfileReport(void)
00109 {
00110     g_u8KACommand = KA_COMMAND_PROFILE_REPORT;
00111 }
00112
00113 //---------------------------------------------------------------------------
00114 void KernelAware::ExitSimulator(void)
00115 {
00116     g_u8KACommand = KA_COMMAND_EXIT_SIMULATOR;
00117 }
00118
00119 //---------------------------------------------------------------------------
00120 void KernelAware::Trace( uint16_t u16File_,
00121                          uint16_t u16Line_ )
00122 {
00123     Trace_i( u16File_, u16Line_, 0, 0, KA_COMMAND_TRACE_0 );
00124 }
00125
00126 //---------------------------------------------------------------------------
```

```
00127 void KernelAware::Trace( uint16_t u16File_,
00128                          uint16_t u16Line_,
00129                          uint16_t u16Arg1_)
00130 {
00131     Trace_i( u16File_, u16Line_, u16Arg1_, 0 ,KA_COMMAND_TRACE_1 );
00132
00133 }
00134 //---------------------------------------------------------------------------
00135 void KernelAware::Trace( uint16_t u16File_,
00136                          uint16_t u16Line_,
00137                          uint16_t u16Arg1_,
00138                          uint16_t u16Arg2_)
00139 {
00140     Trace_i( u16File_, u16Line_, u16Arg1_, u16Arg2_, KA_COMMAND_TRACE_2 );
00141 }
00142
00143 //---------------------------------------------------------------------------
00144 void KernelAware::Trace_i( uint16_t u16File_,
00145                          uint16_t u16Line_,
00146                          uint16_t u16Arg1_,
00147                          uint16_t u16Arg2_,
00148                          KernelAwareCommand_t eCmd_ )
00149 {
00150     CS_ENTER();
00151     g_stKAData.Trace.u16File = u16File_;
00152     g_stKAData.Trace.u16Line = u16Line_;
00153     g_stKAData.Trace.u16Arg1 = u16Arg1_;
00154     g_stKAData.Trace.u16Arg2 = u16Arg2_;
00155     g_u8KACommand = eCmd_;
00156     CS_EXIT();
00157 }
00158
00159 //---------------------------------------------------------------------------
00160 void KernelAware::Print(const char *szStr_)
00161 {
00162     CS_ENTER();
00163     g_stKAData.Print.szString = szStr_;
00164     g_u8KACommand = KA_COMMAND_PRINT;
00165     CS_EXIT();
00166 }
00167
00168 //---------------------------------------------------------------------------
00169 bool KernelAware::IsSimulatorAware(void)
00170 {
00171     return g_bIsKernelAware;
00172 }
00173
00174 #endif
```

15.29 /home/vm/mark3/trunk/embedded/kernel/ksemaphore.cpp File Reference

Semaphore Blocking-Object Implemenation.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ksemaphore.h"
#include "blocking.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
#include "timerlist.h"
```

Functions

- void TimedSemaphore_Callback (Thread *pclOwner_, void *pvData_)

  TimedSemaphore_Callback.

15.29.1 Detailed Description

Semaphore Blocking-Object Implemenation.

Definition in file ksemaphore.cpp.

15.29.2 Function Documentation

15.29.2.1 void TimedSemaphore_Callback (Thread * *pclOwner_*, void * *pvData_*)

TimedSemaphore_Callback.

This function is called from the timer-expired context to trigger a timeout on this semphore. This results in the waking of the thread that generated the semaphore pend call that was not completed in time.

Parameters

*pclOwner_*	Pointer to the thread to wake
*pvData_*	Pointer to the semaphore object that the thread is blocked on

Definition at line 57 of file ksemaphore.cpp.

15.30 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
00028 #define _CAN_HAS_DEBUG
00029 //--[Autogenerated - Do Not Modify]-------------------------------------------
00030 #include "dbg_file_list.h"
00031 #include "buffalogger.h"
00032 #if defined(DBG_FILE)
00033 # error "Debug logging file token already defined!  Bailing."
00034 #else
00035 # define DBG_FILE _DBG___KERNEL_KSEMAPHORE_CPP
00036 #endif
00037 //--[End Autogenerated content]-----------------------------------------------
00038 #include "kerneldebug.h"
00039
00040
00041 #if KERNEL_USE_SEMAPHORE
00042
00043 #if KERNEL_USE_TIMEOUTS
00044 #include "timerlist.h"
00045
00046 //---------------------------------------------------------------------------
00057 void TimedSemaphore_Callback(Thread *pclOwner_, void *pvData_)
00058 {
00059     Semaphore *pclSemaphore = static_cast<Semaphore*>(pvData_);
00060
00061     // Indicate that the semaphore has expired on the thread
00062     pclOwner_->SetExpired(true);
00063
00064     // Wake up the thread that was blocked on this semaphore.
00065     pclSemaphore->WakeMe(pclOwner_);
00066
00067
00068     if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
      ()->GetCurPriority())
00069     {
00070         Thread::Yield();
00071     }
00072 }
00073
```

```
00074 //---------------------------------------------------------------------------
00075 void Semaphore::WakeMe(Thread *pclChosenOne_)
00076 {
00077     // Remove from the semaphore waitlist and back to its ready list.
00078     UnBlock(pclChosenOne_);
00079 }
00080
00081 #endif // KERNEL_USE_TIMEOUTS
00082
00083 //---------------------------------------------------------------------------
00084 uint8_t Semaphore::WakeNext()
00085 {
00086     Thread *pclChosenOne;
00087
00088     pclChosenOne = m_clBlockList.HighestWaiter();
00089
00090     // Remove from the semaphore waitlist and back to its ready list.
00091     UnBlock(pclChosenOne);
00092
00093     // Call a task switch if higher or equal priority thread
00094     if (pclChosenOne->GetCurPriority() >=
      Scheduler::GetCurrentThread()->GetCurPriority())
00095     {
00096         return 1;
00097     }
00098     return 0;
00099 }
00100
00101 //---------------------------------------------------------------------------
00102 void Semaphore::Init(uint16_t u16InitVal_, uint16_t u16MaxVal_)
00103 {
00104     // Copy the paramters into the object - set the maximum value for this
00105     // semaphore to implement either binary or counting semaphores, and set
00106     // the initial count.  Clear the wait list for this object.
00107     m_u16Value = u16InitVal_;
00108     m_u16MaxValue = u16MaxVal_;
00109
00110     m_clBlockList.Init();
00111 }
00112
00113 //---------------------------------------------------------------------------
00114 bool Semaphore::Post()
00115 {
00116     KERNEL_TRACE_1( "Posting semaphore, Thread %d", (uint16_t)g_pclCurrent->
      GetID() );
00117
00118     bool bThreadWake = 0;
00119     bool bBail = false;
00120     // Increment the semaphore count - we can mess with threads so ensure this
00121     // is in a critical section.  We don't just disable the scheudler since
00122     // we want to be able to do this from within an interrupt context as well.
00123     CS_ENTER();
00124
00125     // If nothing is waiting for the semaphore
00126     if (m_clBlockList.GetHead() == NULL)
00127     {
00128         // Check so see if we've reached the maximum value in the semaphore
00129         if (m_u16Value < m_u16MaxValue)
00130         {
00131             // Increment the count value
00132             m_u16Value++;
00133         }
00134         else
00135         {
00136             // Maximum value has been reached, bail out.
00137             bBail = true;
00138         }
00139     }
00140     else
00141     {
00142         // Otherwise, there are threads waiting for the semaphore to be
00143         // posted, so wake the next one (highest priority goes first).
00144         bThreadWake = WakeNext();
00145     }
00146
00147     CS_EXIT();
00148
00149     // If we weren't able to increment the semaphore count, fail out.
00150     if (bBail)
00151     {
00152         return false;
00153     }
00154
00155     // if bThreadWake was set, it means that a higher-priority thread was
00156     // woken.  Trigger a context switch to ensure that this thread gets
00157     // to execute next.
00158     if (bThreadWake)
```

```
00159     {
00160         Thread::Yield();
00161     }
00162     return true;
00163 }
00164
00165 //---------------------------------------------------------------------------
00166 #if KERNEL_USE_TIMEOUTS
00167 bool Semaphore::Pend_i( uint32_t u32WaitTimeMS_ )
00168 #else
00169 void Semaphore::Pend_i( void )
00170 #endif
00171 {
00172     KERNEL_TRACE_1( "Pending semaphore, Thread %d", (uint16_t)g_pclCurrent->
      GetID() );
00173
00174 #if KERNEL_USE_TIMEOUTS
00175     Timer clSemTimer;
00176     bool bUseTimer = false;
00177 #endif
00178
00179     // Once again, messing with thread data - ensure
00180     // we're doing all of these operations from within a thread-safe context.
00181     CS_ENTER();
00182
00183     // Check to see if we need to take any action based on the semaphore count
00184     if (m_u16Value != 0)
00185     {
00186         // The semaphore count is non-zero, we can just decrement the count
00187         // and go along our merry way.
00188         m_u16Value--;
00189     }
00190     else
00191     {
00192         // The semaphore count is zero - we need to block the current thread
00193         // and wait until the semaphore is posted from elsewhere.
00194 #if KERNEL_USE_TIMEOUTS
00195         if (u32WaitTimeMS_)
00196         {
00197             g_pclCurrent->SetExpired(false);
00198             clSemTimer.Init();
00199             clSemTimer.Start(0, u32WaitTimeMS_, TimedSemaphore_Callback, (void*)this
      );
00200             bUseTimer = true;
00201         }
00202 #endif
00203         Block(g_pclCurrent);
00204
00205         // Switch Threads immediately
00206         Thread::Yield();
00207     }
00208
00209     CS_EXIT();
00210
00211 #if KERNEL_USE_TIMEOUTS
00212     if (bUseTimer)
00213     {
00214         clSemTimer.Stop();
00215         return (g_pclCurrent->GetExpired() == 0);
00216     }
00217     return true;
00218 #endif
00219 }
00220
00221 //---------------------------------------------------------------------------
00222 // Redirect the untimed pend API to the timed pend, with a null timeout.
00223 void Semaphore::Pend()
00224 {
00225 #if KERNEL_USE_TIMEOUTS
00226     Pend_i(0);
00227 #else
00228     Pend_i();
00229 #endif
00230 }
00231
00232 #if KERNEL_USE_TIMEOUTS
00233 //---------------------------------------------------------------------------
00234 bool Semaphore::Pend( uint32_t u32WaitTimeMS_ )
00235 {
00236     return Pend_i( u32WaitTimeMS_ );
00237 }
00238 #endif
00239
00240 //---------------------------------------------------------------------------
00241 uint16_t Semaphore::GetCount()
00242 {
00243     uint16_t u16Ret;
```

```
00244     CS_ENTER();
00245     u16Ret = m_u16Value;
00246     CS_EXIT();
00247     return u16Ret;
00248 }
00249
00250 #endif
```

15.31 /home/vm/mark3/trunk/embedded/kernel/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 "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

15.31.1 Detailed Description

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

Definition in file ll.cpp.

15.32 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
00026 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]-------------------------------------
00028 #include "dbg_file_list.h"
00029 #include "buffalogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined!  Bailing."
00032 #else
00033 # define DBG_FILE _DBG___KERNEL_LL_CPP
00034 #endif
00035 //--[End Autogenerated content]-----------------------------------------
00036
00037 #include "kerneldebug.h"
00038
00039 //---------------------------------------------------------------------------
00040 void LinkListNode::ClearNode()
00041 {
00042     next = NULL;
00043     prev = NULL;
00044 }
00045
00046 //---------------------------------------------------------------------------
00047 void DoubleLinkList::Add(LinkListNode *node_)
00048 {
00049     KERNEL_ASSERT( node_ );
00050
00051     // Add a node to the end of the linked list.
00052     if (!m_pstHead)
```

```
00053     {
00054         // If the list is empty, initilize the nodes
00055         m_pstHead = node_;
00056         m_pstTail = node_;
00057 
00058         m_pstHead->prev = NULL;
00059         m_pstTail->next = NULL;
00060         return;
00061     }
00062 
00063     // Move the tail node, and assign it to the new node just passed in
00064     m_pstTail->next = node_;
00065     node_->prev = m_pstTail;
00066     node_->next = NULL;
00067     m_pstTail = node_;
00068 }
00069 
00070 //---------------------------------------------------------------------------
00071 void DoubleLinkList::Remove(LinkListNode *node_)
00072 {
00073     KERNEL_ASSERT( node_ );
00074 
00075     if (node_->prev)
00076     {
00077 #if SAFE_UNLINK
00078         if (node_->prev->next != node_)
00079         {
00080             Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00081         }
00082 #endif
00083         node_->prev->next = node_->next;
00084     }
00085     if (node_->next)
00086     {
00087 #if SAFE_UNLINK
00088         if (node_->next->prev != node_)
00089         {
00090             Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00091         }
00092 #endif
00093         node_->next->prev = node_->prev;
00094     }
00095     if (node_ == m_pstHead)
00096     {
00097         m_pstHead = node_->next;
00098     }
00099     if (node_ == m_pstTail)
00100     {
00101         m_pstTail = node_->prev;
00102     }
00103 
00104     node_->ClearNode();
00105 }
00106 
00107 //---------------------------------------------------------------------------
00108 void CircularLinkList::Add(LinkListNode *node_)
00109 {
00110     KERNEL_ASSERT( node_ );
00111 
00112     // Add a node to the end of the linked list.
00113     if (!m_pstHead)
00114     {
00115         // If the list is empty, initilize the nodes
00116         m_pstHead = node_;
00117         m_pstTail = node_;
00118 
00119         m_pstHead->prev = m_pstHead;
00120         m_pstHead->next = m_pstHead;
00121         return;
00122     }
00123 
00124     // Move the tail node, and assign it to the new node just passed in
00125     m_pstTail->next = node_;
00126     node_->prev = m_pstTail;
00127     node_->next = m_pstHead;
00128     m_pstTail = node_;
00129     m_pstHead->prev = node_;
00130 }
00131 
00132 //---------------------------------------------------------------------------
00133 void CircularLinkList::Remove(LinkListNode *node_)
00134 {
00135     KERNEL_ASSERT( node_ );
00136 
00137     // Check to see if this is the head of the list...
00138     if ((node_ == m_pstHead) && (m_pstHead == m_pstTail))
00139     {
```

```
00140         // Clear the head and tail pointers - nothing else left.
00141         m_pstHead = NULL;
00142         m_pstTail = NULL;
00143         return;
00144     }
00145
00146 #if SAFE_UNLINK
00147     // Verify that all nodes are properly connected
00148     if ((node_->prev->next != node_) || (node_->next->prev != node_))
00149     {
00150         Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00151     }
00152 #endif
00153
00154     // This is a circularly linked list - no need to check for connection,
00155     // just remove the node.
00156     node_->next->prev = node_->prev;
00157     node_->prev->next = node_->next;
00158
00159     if (node_ == m_pstHead)
00160     {
00161         m_pstHead = m_pstHead->next;
00162     }
00163     if (node_ == m_pstTail)
00164     {
00165         m_pstTail = m_pstTail->prev;
00166     }
00167     node_->ClearNode();
00168 }
00169
00170 //---------------------------------------------------------------------------
00171 void CircularLinkList::PivotForward()
00172 {
00173     if (m_pstHead)
00174     {
00175         m_pstHead = m_pstHead->next;
00176         m_pstTail = m_pstTail->next;
00177     }
00178 }
00179
00180 //---------------------------------------------------------------------------
00181 void CircularLinkList::PivotBackward()
00182 {
00183     if (m_pstHead)
00184     {
00185         m_pstHead = m_pstHead->prev;
00186         m_pstTail = m_pstTail->prev;
00187     }
00188 }
```

15.33 /home/vm/mark3/trunk/embedded/kernel/mailbox.cpp File Reference

Mailbox + Envelope IPC mechanism.

```
#include "mark3cfg.h"
#include "kerneltypes.h"
#include "ksemaphore.h"
#include "mailbox.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

15.33.1 Detailed Description

Mailbox + Envelope IPC mechanism.

Definition in file mailbox.cpp.

15.34 mailbox.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 "mark3cfg.h"
00022 #include "kerneltypes.h"
00023 #include "ksemaphore.h"
00024 #include "mailbox.h"
00025
00026 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]-------------------------------------------
00028 #include "dbg_file_list.h"
00029 #include "buffalogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined!  Bailing."
00032 #else
00033 # define DBG_FILE _DBG___KERNEL_MAILBOX_CPP
00034 #endif
00035 //--[End Autogenerated content]-----------------------------------------------
00036
00037 #include "kerneldebug.h"
00038
00039 #if KERNEL_USE_MAILBOX
00040
00041 //---------------------------------------------------------------------------
00042 void Mailbox::Init( void *pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_ )
00043 {
00044     KERNEL_ASSERT(u16BufferSize_);
00045     KERNEL_ASSERT(u16ElementSize_);
00046     KERNEL_ASSERT(pvBuffer_);
00047
00048     m_pvBuffer = pvBuffer_;
00049     m_uselementSize = u16ElementSize_;
00050
00051     m_u16Count = (u16BufferSize_ / u16ElementSize_);
00052     m_u16Free = m_u16Count;
00053
00054     m_u16Head = 0;
00055     m_u16Tail = 0;
00056
00057     // We use the counting semaphore to implement blocking - with one element
00058     // in the mailbox corresponding to a post/pend operation in the semaphore.
00059     m_clRecvSem.Init(0, m_u16Free);
00060
00061 #if KERNEL_USE_TIMEOUTS
00062     // Binary semaphore is used to track any threads that are blocked on a
00063     // "send" due to lack of free slots.
00064     m_clSendSem.Init(0, 1);
00065 #endif
00066 }
00067
00068 //---------------------------------------------------------------------------
00069 void Mailbox::Receive( void *pvData_ )
00070 {
00071     KERNEL_ASSERT( pvData_ );
00072
00073 #if KERNEL_USE_TIMEOUTS
00074     Receive_i( pvData_, false, 0 );
00075 #else
00076     Receive_i( pvData_, false );
00077 #endif
00078 }
00079
00080 #if KERNEL_USE_TIMEOUTS
00081 //---------------------------------------------------------------------------
00082 bool Mailbox::Receive( void *pvData_, uint32_t u32TimeoutMS_ )
00083 {
00084     KERNEL_ASSERT( pvData_ );
00085     return Receive_i( pvData_, false, u32TimeoutMS_ );
00086 }
00087 #endif
00088
00089 //---------------------------------------------------------------------------
00090 void Mailbox::ReceiveTail( void *pvData_ )
00091 {
```

```
00092     KERNEL_ASSERT( pvData_ );
00093
00094 #if KERNEL_USE_TIMEOUTS
00095     Receive_i( pvData_, true, 0 );
00096 #else
00097     Receive_i( pvData_, true );
00098 #endif
00099 }
00100
00101 #if KERNEL_USE_TIMEOUTS
00102 //---------------------------------------------------------------------------
00103 bool Mailbox::ReceiveTail( void *pvData_, uint32_t u32TimeoutMS_ )
00104 {
00105     KERNEL_ASSERT( pvData_ );
00106     return Receive_i( pvData_, true, u32TimeoutMS_ );
00107 }
00108 #endif
00109
00110 //---------------------------------------------------------------------------
00111 bool Mailbox::Send( void *pvData_ )
00112 {
00113     KERNEL_ASSERT( pvData_ );
00114
00115 #if KERNEL_USE_TIMEOUTS
00116     return Send_i( pvData_, false, 0 );
00117 #else
00118     return Send_i( pvData_, false );
00119 #endif
00120 }
00121
00122 //---------------------------------------------------------------------------
00123 bool Mailbox::SendTail( void *pvData_ )
00124 {
00125     KERNEL_ASSERT( pvData_ );
00126
00127 #if KERNEL_USE_TIMEOUTS
00128     return Send_i( pvData_, true, 0 );
00129 #else
00130     return Send_i( pvData_, true );
00131 #endif
00132 }
00133
00134 #if KERNEL_USE_TIMEOUTS
00135 //---------------------------------------------------------------------------
00136 bool Mailbox::Send( void *pvData_, uint32_t u32TimeoutMS_ )
00137 {
00138     KERNEL_ASSERT( pvData_ );
00139
00140     return Send_i( pvData_, false, u32TimeoutMS_ );
00141 }
00142
00143 //---------------------------------------------------------------------------
00144 bool Mailbox::SendTail( void *pvData_, uint32_t u32TimeoutMS_ )
00145 {
00146     KERNEL_ASSERT( pvData_ );
00147
00148     return Send_i( pvData_, true, u32TimeoutMS_ );
00149 }
00150 #endif
00151
00152 //---------------------------------------------------------------------------
00153 #if KERNEL_USE_TIMEOUTS
00154 bool Mailbox::Send_i( const void *pvData_, bool bTail_, uint32_t u32TimeoutMS_)
00155 #else
00156 bool Mailbox::Send_i( const void *pvData_, bool bTail_)
00157 #endif
00158 {
00159     const void *pvDst;
00160
00161     bool bRet = false;
00162     bool bSchedState = Scheduler::SetScheduler( false );
00163
00164 #if KERNEL_USE_TIMEOUTS
00165     bool bBlock = false;
00166     bool bDone = false;
00167     while (!bDone)
00168     {
00169         // Try to claim a slot first before resorting to blocking.
00170         if (bBlock)
00171         {
00172             bDone = true;
00173             Scheduler::SetScheduler( bSchedState );
00174             m_clSendSem.Pend( u32TimeoutMS_ );
00175             Scheduler::SetScheduler( false );
00176         }
00177 #endif
00178
```

```
00179         CS_ENTER();
00180         // Ensure we have a free slot before we attempt to write data
00181         if (m_u16Free)
00182         {
00183             m_u16Free--;
00184 
00185             if (bTail_)
00186             {
00187                 pvDst = GetTailPointer();
00188                 MoveTailBackward();
00189             }
00190             else
00191             {
00192                 MoveHeadForward();
00193                 pvDst = GetHeadPointer();
00194             }
00195             bRet = true;
00196 #if KERNEL_USE_TIMEOUTS
00197             bDone = true;
00198 #endif
00199         }
00200 
00201 #if KERNEL_USE_TIMEOUTS
00202         else if (u32TimeoutMS_)
00203         {
00204             bBlock = true;
00205         }
00206         else
00207         {
00208             bDone = true;
00209         }
00210 #endif
00211 
00212         CS_EXIT();
00213 
00214 #if KERNEL_USE_TIMEOUTS
00215     }
00216 #endif
00217 
00218     // Copy data to the claimed slot, and post the counting semaphore
00219     if (bRet)
00220     {
00221         CopyData( pvData_, pvDst, m_uselementSize );
00222     }
00223 
00224     Scheduler::SetScheduler( bSchedState );
00225 
00226     if (bRet)
00227     {
00228         m_clRecvSem.Post();
00229     }
00230 
00231     return bRet;
00232 }
00233 
00234 //---------------------------------------------------------------------------
00235 #if KERNEL_USE_TIMEOUTS
00236 bool Mailbox::Receive_i( const void *pvData_, bool bTail_, uint32_t u32WaitTimeMS_ )
00237 #else
00238 void Mailbox::Receive_i( const void *pvData_, bool bTail_ )
00239 #endif
00240 {
00241     const void *pvSrc;
00242 
00243 #if KERNEL_USE_TIMEOUTS
00244     if (!m_clRecvSem.Pend( u32WaitTimeMS_ ))
00245     {
00246         // Failed to get the notification from the counting semaphore in the
00247         // time allotted.  Bail.
00248         return false;
00249     }
00250 #else
00251     m_clRecvSem.Pend();
00252 #endif
00253 
00254     // Disable the scheduler while we do this -- this ensures we don't have
00255     // multiple concurrent readers off the same queue, which could be problematic
00256     // if multiple writes occur during reads, etc.
00257     bool bSchedState = Scheduler::SetScheduler( false );
00258 
00259     // Update the head/tail indexes, and get the associated data pointer for
00260     // the read operation.
00261     CS_ENTER();
00262 
00263     m_u16Free++;
00264     if (bTail_)
00265     {
```

```
00266         MoveTailForward();
00267         pvSrc = GetTailPointer();
00268     }
00269     else
00270     {
00271         pvSrc = GetHeadPointer();
00272         MoveHeadBackward();
00273     }
00274 
00275     CS_EXIT();
00276 
00277     CopyData( pvSrc, pvData_, m_uselementSize );
00278 
00279     Scheduler::SetScheduler( bSchedState );
00280 
00281     // Unblock a thread waiting for a free slot to send to
00282     m_clSendSem.Post();
00283 
00284 #if KERNEL_USE_TIMEOUTS
00285     return true;
00286 #endif
00287 }
00288 
00289 #endif
```

15.35 /home/vm/mark3/trunk/embedded/kernel/message.cpp File Reference

Inter-thread communications via message passing.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "message.h"
#include "threadport.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
#include "timerlist.h"
```

15.35.1 Detailed Description

Inter-thread communications via message passing.

Definition in file message.cpp.

15.36 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 
00028 #define _CAN_HAS_DEBUG
00029 //--[Autogenerated - Do Not Modify]-------------------------------------
00030 #include "dbg_file_list.h"
00031 #include "buffalogger.h"
00032 #if defined(DBG_FILE)
```

```
00033 # error "Debug logging file token already defined!  Bailing."
00034 #else
00035 # define DBG_FILE _DBG___KERNEL_MESSAGE_CPP
00036 #endif
00037 //--[End Autogenerated content]-----------------------------------------
00038 #include "kerneldebug.h"
00039
00040 #if KERNEL_USE_MESSAGE
00041
00042 #if KERNEL_USE_TIMEOUTS
00043     #include "timerlist.h"
00044 #endif
00045
00046 Message GlobalMessagePool::m_aclMessagePool[
      GLOBAL_MESSAGE_POOL_SIZE];
00047 DoubleLinkList GlobalMessagePool::m_clList;
00048
00049 //---------------------------------------------------------------------------
00050 void GlobalMessagePool::Init()
00051 {
00052     uint8_t i;
00053     GlobalMessagePool::m_clList.Init();
00054     for (i = 0; i < GLOBAL_MESSAGE_POOL_SIZE; i++)
00055     {
00056         GlobalMessagePool::m_aclMessagePool[i].Init();
00057         GlobalMessagePool::m_clList.Add(&(GlobalMessagePool::m_aclMessagePool[i]));
00058     }
00059 }
00060
00061 //---------------------------------------------------------------------------
00062 void GlobalMessagePool::Push( Message *pclMessage_ )
00063 {
00064     KERNEL_ASSERT( pclMessage_ );
00065
00066     CS_ENTER();
00067
00068     GlobalMessagePool::m_clList.Add(pclMessage_);
00069
00070     CS_EXIT();
00071 }
00072
00073 //---------------------------------------------------------------------------
00074 Message *GlobalMessagePool::Pop()
00075 {
00076     Message *pclRet;
00077     CS_ENTER();
00078
00079     pclRet = static_cast<Message*>( GlobalMessagePool::m_clList.GetHead() );
00080     if (0 != pclRet)
00081     {
00082         GlobalMessagePool::m_clList.Remove( static_cast<LinkListNode*>( pclRet ) );
00083     }
00084
00085     CS_EXIT();
00086     return pclRet;
00087 }
00088
00089 //---------------------------------------------------------------------------
00090 void MessageQueue::Init()
00091 {
00092     m_clSemaphore.Init(0, GLOBAL_MESSAGE_POOL_SIZE);
00093 }
00094
00095 //---------------------------------------------------------------------------
00096 Message *MessageQueue::Receive()
00097 {
00098 #if KERNEL_USE_TIMEOUTS
00099     return Receive_i(0);
00100 #else
00101     return Receive_i();
00102 #endif
00103 }
00104
00105 //---------------------------------------------------------------------------
00106 #if KERNEL_USE_TIMEOUTS
00107 Message *MessageQueue::Receive( uint32_t u32TimeWaitMS_)
00108 {
00109     return Receive_i( u32TimeWaitMS_ );
00110 }
00111 #endif
00112
00113 //---------------------------------------------------------------------------
00114 #if KERNEL_USE_TIMEOUTS
00115 Message *MessageQueue::Receive_i( uint32_t u32TimeWaitMS_ )
00116 #else
00117 Message *MessageQueue::Receive_i( void )
00118 #endif
```

```
00119 {
00120     Message *pclRet;
00121 
00122     // Block the current thread on the counting semaphore
00123 #if KERNEL_USE_TIMEOUTS
00124     if (!m_clSemaphore.Pend(u32TimeWaitMS_))
00125     {
00126         return NULL;
00127     }
00128 #else
00129     m_clSemaphore.Pend();
00130 #endif
00131 
00132     CS_ENTER();
00133 
00134     // Pop the head of the message queue and return it
00135     pclRet = static_cast<Message*>( m_clLinkList.GetHead() );
00136     m_clLinkList.Remove(static_cast<Message*>(pclRet));
00137 
00138     CS_EXIT();
00139 
00140     return pclRet;
00141 }
00142 
00143 //---------------------------------------------------------------------------
00144 void MessageQueue::Send( Message *pclSrc_ )
00145 {
00146     KERNEL_ASSERT( pclSrc_ );
00147 
00148     CS_ENTER();
00149 
00150     // Add the message to the head of the linked list
00151     m_clLinkList.Add( pclSrc_ );
00152 
00153     // Post the semaphore, waking the blocking thread for the queue.
00154     m_clSemaphore.Post();
00155 
00156     CS_EXIT();
00157 }
00158 
00159 //---------------------------------------------------------------------------
00160 uint16_t MessageQueue::GetCount()
00161 {
00162     return m_clSemaphore.GetCount();
00163 }
00164 #endif //KERNEL_USE_MESSAGE
```

15.37 /home/vm/mark3/trunk/embedded/kernel/mutex.cpp File Reference

Mutual-exclusion object.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "blocking.h"
#include "mutex.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

Functions

- void TimedMutex_Calback (Thread *pclOwner_, void *pvData_)

  TimedMutex_Calback.

15.37.1 Detailed Description

Mutual-exclusion object.

Definition in file mutex.cpp.

15.37.2 Function Documentation

15.37.2.1 void TimedMutex_Calback (Thread * *pclOwner_*, void * *pvData_*)

TimedMutex_Calback.

This function is called from the timer-expired context to trigger a timeout on this mutex. This results in the waking of the thread that generated the mutex claim call that was not completed in time.

Parameters

*pclOwner_*	Pointer to the thread to wake
*pvData_*	Pointer to the mutex object that the thread is blocked on

Definition at line 54 of file mutex.cpp.

15.38 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
00026 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]-------------------------------------
00028 #include "dbg_file_list.h"
00029 #include "buffalogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined!  Bailing."
00032 #else
00033 # define DBG_FILE _DBG___KERNEL_MUTEX_CPP
00034 #endif
00035 //--[End Autogenerated content]-----------------------------------------
00036
00037 #include "kerneldebug.h"
00038
00039 #if KERNEL_USE_MUTEX
00040
00041 #if KERNEL_USE_TIMEOUTS
00042
00043 //---------------------------------------------------------------------------
00054 void TimedMutex_Calback(Thread *pclOwner_, void *pvData_)
00055 {
00056     Mutex *pclMutex = static_cast<Mutex*>(pvData_);
00057
00058     // Indicate that the semaphore has expired on the thread
00059     pclOwner_->SetExpired(true);
00060
00061     // Wake up the thread that was blocked on this semaphore.
00062     pclMutex->WakeMe(pclOwner_);
00063
00064     if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
      ()->GetCurPriority())
00065     {
00066         Thread::Yield();
00067     }
00068 }
00069
00070 //---------------------------------------------------------------------------
00071 void Mutex::WakeMe(Thread *pclOwner_)
00072 {
00073     // Remove from the semaphore waitlist and back to its ready list.
00074     UnBlock(pclOwner_);
00075 }
```

```
00076
00077 #endif
00078
00079 //---------------------------------------------------------------------------
00080 uint8_t Mutex::WakeNext()
00081 {
00082     Thread *pclChosenOne = NULL;
00083
00084     // Get the highest priority waiter thread
00085     pclChosenOne = m_clBlockList.HighestWaiter();
00086
00087     // Unblock the thread
00088     UnBlock(pclChosenOne);
00089
00090     // The chosen one now owns the mutex
00091     m_pclOwner = pclChosenOne;
00092
00093     // Signal a context switch if it's a greater than or equal to the current priority
00094     if (pclChosenOne->GetCurPriority() >=
      Scheduler::GetCurrentThread()->GetCurPriority())
00095     {
00096         return 1;
00097     }
00098     return 0;
00099 }
00100
00101 //---------------------------------------------------------------------------
00102 void Mutex::Init()
00103 {
00104     // Reset the data in the mutex
00105     m_bReady = 1;              // The mutex is free.
00106     m_u8MaxPri = 0;            // Set the maximum priority inheritence state
00107     m_pclOwner = NULL;         // Clear the mutex owner
00108     m_u8Recurse = 0;           // Reset recurse count
00109 }
00110
00111 //---------------------------------------------------------------------------
00112 #if KERNEL_USE_TIMEOUTS
00113 bool Mutex::Claim_i(uint32_t u32WaitTimeMS_)
00114 #else
00115 void Mutex::Claim_i(void)
00116 #endif
00117 {
00118     KERNEL_TRACE_1( "Claiming Mutex, Thread %d", (uint16_t)g_pclCurrent->
      GetID() );
00119
00120 #if KERNEL_USE_TIMEOUTS
00121     Timer clTimer;
00122     bool bUseTimer = false;
00123 #endif
00124
00125     // Disable the scheduler while claiming the mutex - we're dealing with all
00126     // sorts of private thread data, can't have a thread switch while messing
00127     // with internal data structures.
00128     Scheduler::SetScheduler(0);
00129
00130     // Check to see if the mutex is claimed or not
00131     if (m_bReady != 0)
00132     {
00133         // Mutex isn't claimed, claim it.
00134         m_bReady = 0;
00135         m_u8Recurse = 0;
00136         m_u8MaxPri = g_pclCurrent->GetPriority();
00137         m_pclOwner = g_pclCurrent;
00138
00139         Scheduler::SetScheduler(1);
00140
00141 #if KERNEL_USE_TIMEOUTS
00142         return true;
00143 #else
00144         return;
00145 #endif
00146     }
00147
00148     // If the mutex is already claimed, check to see if this is the owner thread,
00149     // since we allow the mutex to be claimed recursively.
00150     if (g_pclCurrent == m_pclOwner)
00151     {
00152         // Ensure that we haven't exceeded the maximum recursive-lock count
00153         KERNEL_ASSERT( (m_u8Recurse < 255) );
00154         m_u8Recurse++;
00155
00156         // Increment the lock count and bail
00157         Scheduler::SetScheduler(1);
00158 #if KERNEL_USE_TIMEOUTS
00159         return true;
00160 #else
```

```
00161         return;
00162 #endif
00163     }
00164 
00165     // The mutex is claimed already - we have to block now.  Move the
00166     // current thread to the list of threads waiting on the mutex.
00167 #if KERNEL_USE_TIMEOUTS
00168     if (u32WaitTimeMS_)
00169     {
00170         g_pclCurrent->SetExpired(false);
00171         clTimer.Init();
00172         clTimer.Start(0, u32WaitTimeMS_, (TimerCallback_t)
      TimedMutex_Calback, (void*)this);
00173         bUseTimer = true;
00174     }
00175 #endif
00176     Block(g_pclCurrent);
00177 
00178     // Check if priority inheritence is necessary.  We do this in order
00179     // to ensure that we don't end up with priority inversions in case
00180     // multiple threads are waiting on the same resource.
00181     if(m_u8MaxPri <= g_pclCurrent->GetPriority())
00182     {
00183         m_u8MaxPri = g_pclCurrent->GetPriority();
00184 
00185         Thread *pclTemp = static_cast<Thread*>(m_clBlockList.GetHead());
00186         while(pclTemp)
00187         {
00188             pclTemp->InheritPriority(m_u8MaxPri);
00189             if(pclTemp == static_cast<Thread*>(m_clBlockList.GetTail()) )
00190             {
00191                 break;
00192             }
00193             pclTemp = static_cast<Thread*>(pclTemp->GetNext());
00194         }
00195         m_pclOwner->InheritPriority(m_u8MaxPri);
00196     }
00197 
00198     // Done with thread data -reenable the scheduler
00199     Scheduler::SetScheduler(1);
00200 
00201     // Switch threads if this thread acquired the mutex
00202     Thread::Yield();
00203 
00204 #if KERNEL_USE_TIMEOUTS
00205     if (bUseTimer)
00206     {
00207         clTimer.Stop();
00208         return (g_pclCurrent->GetExpired() == 0);
00209     }
00210     return true;
00211 #endif
00212 }
00213 
00214 //---------------------------------------------------------------------------
00215 void Mutex::Claim(void)
00216 {
00217 #if KERNEL_USE_TIMEOUTS
00218     Claim_i(0);
00219 #else
00220     Claim_i();
00221 #endif
00222 }
00223 
00224 //---------------------------------------------------------------------------
00225 #if KERNEL_USE_TIMEOUTS
00226 bool Mutex::Claim(uint32_t u32WaitTimeMS_)
00227 {
00228     return Claim_i(u32WaitTimeMS_);
00229 }
00230 #endif
00231 
00232 //---------------------------------------------------------------------------
00233 void Mutex::Release()
00234 {
00235     KERNEL_TRACE_1( "Releasing Mutex, Thread %d", (uint16_t)g_pclCurrent->
      GetID() );
00236 
00237     bool bSchedule = 0;
00238 
00239     // Disable the scheduler while we deal with internal data structures.
00240     Scheduler::SetScheduler(0);
00241 
00242     // This thread had better be the one that owns the mutex currently...
00243     KERNEL_ASSERT( (g_pclCurrent == m_pclOwner) );
00244 
00245     // If the owner had claimed the lock multiple times, decrease the lock
```

```
00246     // count and return immediately.
00247     if (m_u8Recurse)
00248     {
00249         m_u8Recurse--;
00250         Scheduler::SetScheduler(1);
00251         return;
00252     }
00253 
00254     // Restore the thread's original priority
00255     if (g_pclCurrent->GetCurPriority() != g_pclCurrent->
      GetPriority())
00256     {
00257         g_pclCurrent->SetPriority(g_pclCurrent->
      GetPriority());
00258 
00259         // In this case, we want to reschedule
00260         bSchedule = 1;
00261     }
00262 
00263     // No threads are waiting on this semaphore?
00264     if (m_clBlockList.GetHead() == NULL)
00265     {
00266         // Re-initialize the mutex to its default values
00267         m_bReady = 1;
00268         m_u8MaxPri = 0;
00269         m_pclOwner = NULL;
00270     }
00271     else
00272     {
00273         // Wake the highest priority Thread pending on the mutex
00274         if(WakeNext())
00275         {
00276             // Switch threads if it's higher or equal priority than the current thread
00277             bSchedule = 1;
00278         }
00279     }
00280 
00281     // Must enable the scheduler again in order to switch threads.
00282     Scheduler::SetScheduler(1);
00283     if(bSchedule)
00284     {
00285         // Switch threads if a higher-priority thread was woken
00286         Thread::Yield();
00287     }
00288 }
00289 
00290 #endif //KERNEL_USE_MUTEX
```

15.39 /home/vm/mark3/trunk/embedded/kernel/notify.cpp File Reference

Lightweight thread notification - blocking object.

```
#include "mark3cfg.h"
#include "notify.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
```

15.39.1 Detailed Description

Lightweight thread notification - blocking object.

Definition in file notify.cpp.

15.40 notify.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 ===========================================================================*/
00023 #include "mark3cfg.h"
00024 #include "notify.h"
00025
00026 #define _CAN_HAS_DEBUG
00027 //--[Autogenerated - Do Not Modify]---------------------------------------
00028 #include "dbg_file_list.h"
00029 #include "buffalogger.h"
00030 #if defined(DBG_FILE)
00031 # error "Debug logging file token already defined!  Bailing."
00032 #else
00033 # define DBG_FILE _DBG___KERNEL_NOTIFY_CPP
00034 #endif
00035 //--[End Autogenerated content]-------------------------------------------
00036
00037 #if KERNEL_USE_NOTIFY
00038 //---------------------------------------------------------------------------
00039 void TimedNotify_Callback(Thread *pclOwner_, void *pvData_)
00040 {
00041     Notify *pclNotify = static_cast<Notify*>(pvData_);
00042
00043     // Indicate that the semaphore has expired on the thread
00044     pclOwner_->SetExpired(true);
00045
00046     // Wake up the thread that was blocked on this semaphore.
00047     pclNotify->WakeMe(pclOwner_);
00048
00049     if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
      ()->GetCurPriority())
00050     {
00051         Thread::Yield();
00052     }
00053 }
00054
00055 //---------------------------------------------------------------------------
00056 void Notify::Init(void)
00057 {
00058     m_clBlockList.Init();
00059 }
00060
00061 //---------------------------------------------------------------------------
00062 void Notify::Signal(void)
00063 {
00064     bool bReschedule = false;
00065
00066     CS_ENTER();
00067     Thread *pclCurrent = (Thread*)m_clBlockList.GetHead();
00068     while (pclCurrent != NULL)
00069     {
00070         UnBlock(pclCurrent);
00071         if ( !bReschedule &&
00072            ( pclCurrent->GetCurPriority() >=
      Scheduler::GetCurrentThread()->GetCurPriority() ) )
00073         {
00074             bReschedule = true;
00075         }
00076         pclCurrent = (Thread*)m_clBlockList.GetHead();
00077     }
00078     CS_EXIT();
00079
00080     if (bReschedule)
00081     {
00082         Thread::Yield();
00083     }
00084 }
00085
00086 //---------------------------------------------------------------------------
00087 void Notify::Wait( bool *pbFlag_ )
00088 {
00089     CS_ENTER();
00090     Block(g_pclCurrent);
00091     if (pbFlag_)
00092     {
00093         *pbFlag_ = false;
00094     }
00095     CS_EXIT();
00096
00097     Thread::Yield();
00098     if (pbFlag_)
00099     {
00100         *pbFlag_ = true;
00101     }
```

```
00102 }
00103
00104 //---------------------------------------------------------------------------
00105 #if KERNEL_USE_TIMEOUTS
00106 bool Notify::Wait( uint32_t u32WaitTimeMS_, bool *pbFlag_ )
00107 {
00108     bool bUseTimer = false;
00109     Timer clNotifyTimer;
00110
00111     CS_ENTER();
00112     if (u32WaitTimeMS_)
00113     {
00114         bUseTimer = true;
00115         g_pclCurrent->SetExpired(false);
00116
00117         clNotifyTimer.Init();
00118         clNotifyTimer.Start(0, u32WaitTimeMS_, TimedNotify_Callback, (void*)this);
00119     }
00120
00121     Block(g_pclCurrent);
00122
00123     if (pbFlag_)
00124     {
00125         *pbFlag_ = false;
00126     }
00127     CS_EXIT();
00128
00129     Thread::Yield();
00130
00131     if (bUseTimer)
00132     {
00133         clNotifyTimer.Stop();
00134         return (g_pclCurrent->GetExpired() == 0);
00135     }
00136
00137     if (pbFlag_)
00138     {
00139         *pbFlag_ = true;
00140     }
00141
00142     return true;
00143 }
00144 #endif
00145 //---------------------------------------------------------------------------
00146 void Notify::WakeMe(Thread *pclChosenOne_)
00147 {
00148     UnBlock(pclChosenOne_);
00149 }
00150
00151 #endif
```

15.41 /home/vm/mark3/trunk/embedded/kernel/profile.cpp File Reference

Code profiling utilities.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "profile.h"
#include "kernelprofile.h"
#include "threadport.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

15.41.1 Detailed Description

Code profiling utilities.

Definition in file profile.cpp.

15.42 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 "kernelprofile.h"
00025 #include "threadport.h"
00026
00027 #define _CAN_HAS_DEBUG
00028 //--[Autogenerated - Do Not Modify]-------------------------------------------
00029 #include "dbg_file_list.h"
00030 #include "buffalogger.h"
00031 #if defined(DBG_FILE)
00032 # error "Debug logging file token already defined!  Bailing."
00033 #else
00034 # define DBG_FILE _DBG___KERNEL_PROFILE_CPP
00035 #endif
00036 //--[End Autogenerated content]-----------------------------------------------
00037
00038 #include "kerneldebug.h"
00039
00040 #if KERNEL_USE_PROFILER
00041
00042 //---------------------------------------------------------------------------
00043 void ProfileTimer::Init()
00044 {
00045     m_u32Cumulative = 0;
00046     m_u32CurrentIteration = 0;
00047     m_u16Iterations = 0;
00048     m_bActive = 0;
00049 }
00050
00051 //---------------------------------------------------------------------------
00052 void ProfileTimer::Start()
00053 {
00054     if (!m_bActive)
00055     {
00056         CS_ENTER();
00057         m_u32CurrentIteration = 0;
00058         m_u32InitialEpoch = Profiler::GetEpoch();
00059         m_u16Initial = Profiler::Read();
00060         CS_EXIT();
00061         m_bActive = 1;
00062     }
00063 }
00064
00065 //---------------------------------------------------------------------------
00066 void ProfileTimer::Stop()
00067 {
00068     if (m_bActive)
00069     {
00070         uint16_t u16Final;
00071         uint32_t u32Epoch;
00072         CS_ENTER();
00073         u16Final = Profiler::Read();
00074         u32Epoch = Profiler::GetEpoch();
00075         // Compute total for current iteration...
00076         m_u32CurrentIteration = ComputeCurrentTicks(u16Final,
      u32Epoch);
00077         m_u32Cumulative += m_u32CurrentIteration;
00078         m_u16Iterations++;
00079         CS_EXIT();
00080         m_bActive = 0;
00081     }
00082 }
00083
00084 //---------------------------------------------------------------------------
00085 uint32_t ProfileTimer::GetAverage()
00086 {
00087     if (m_u16Iterations)
00088     {
00089         return m_u32Cumulative / (uint32_t)m_u16Iterations;
00090     }
```

```
00091     return 0;
00092 }
00093
00094 //---------------------------------------------------------------------------
00095 uint32_t ProfileTimer::GetCurrent()
00096 {
00097
00098     if (m_bActive)
00099     {
00100         uint16_t u16Current;
00101         uint32_t u32Epoch;
00102         CS_ENTER();
00103         u16Current = Profiler::Read();
00104         u32Epoch = Profiler::GetEpoch();
00105         CS_EXIT();
00106         return ComputeCurrentTicks(u16Current, u32Epoch);
00107     }
00108     return m_u32CurrentIteration;
00109 }
00110
00111 //---------------------------------------------------------------------------
00112 uint32_t ProfileTimer::ComputeCurrentTicks(uint16_t u16Current_, uint32_t
      u32Epoch_)
00113 {
00114     uint32_t u32Total;
00115     uint32_t u32Overflows;
00116
00117     u32Overflows = u32Epoch_ - m_u32InitialEpoch;
00118
00119     // More than one overflow...
00120     if (u32Overflows > 1)
00121     {
00122         u32Total = ((uint32_t)(u32Overflows-1) * TICKS_PER_OVERFLOW)
00123                 + (uint32_t)(TICKS_PER_OVERFLOW - m_u16Initial) +
00124                 (uint32_t)u16Current_;
00125     }
00126     // Only one overflow, or one overflow that has yet to be processed
00127     else if (u32Overflows || (u16Current_ < m_u16Initial))
00128     {
00129         u32Total = (uint32_t)(TICKS_PER_OVERFLOW - m_u16Initial) +
00130                 (uint32_t)u16Current_;
00131     }
00132     // No overflows, none pending.
00133     else
00134     {
00135         u32Total = (uint32_t)(u16Current_ - m_u16Initial);
00136     }
00137
00138     return u32Total;
00139 }
00140
00141 #endif
```

15.43 /home/vm/mark3/trunk/embedded/kernel/public/atomic.h File Reference

Basic Atomic Operations.

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

15.43.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.h.

15.44 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 uint8_t Set( uint8_t *pu8Source_, uint8_t u8Val_ );
00049     static uint16_t Set( uint16_t *pu16Source_, uint16_t u16Val_ );
00050     static uint32_t Set( uint32_t *pu32Source_, uint32_t u32Val_ );
00051
00058     static uint8_t Add( uint8_t *pu8Source_, uint8_t u8Val_ );
00059     static uint16_t Add( uint16_t *pu16Source_, uint16_t u16Val_ );
00060     static uint32_t Add( uint32_t *pu32Source_, uint32_t u32Val_ );
00061
00068     static uint8_t Sub( uint8_t *pu8Source_, uint8_t u8Val_ );
00069     static uint16_t Sub( uint16_t *pu16Source_, uint16_t u16Val_ );
00070     static uint32_t Sub( uint32_t *pu32Source_, uint32_t u32Val_ );
00071
00086     static bool TestAndSet( bool *pbLock );
00087 };
00088
00089 #endif // KERNEL_USE_ATOMIC
00090
00091 #endif //__ATOMIC_H__
```

15.45 /home/vm/mark3/trunk/embedded/kernel/public/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.

15.45.1 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 manipu32ate 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.

15.46 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
```

15.47 /home/vm/mark3/trunk/embedded/kernel/public/buffalogger.h File Reference

Super-efficient, super-secure logging routines.

`#include <stdint.h>`

15.47.1 Detailed Description

Super-efficient, super-secure logging routines.

Uses offline processing to ensure performance.

Definition in file buffalogger.h.

15.48 buffalogger.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 #pragma once
00021 #include <stdint.h>
00022
00023 //---------------------------------------------------------------------------
00024 #define STR1(s) #s
00025 #define STR(s) STR1(s)
00026
00027 //---------------------------------------------------------------------------
00028 #define EMIT_DBG_STRING(str) \
00029 { \
00030     const static volatile char log_str[] __attribute__((section (".logger"))) = str; \
00031     const static volatile uint16_t line_id __attribute__((section (".logger"))) = __LINE__; \
00032     const static volatile uint16_t file_id __attribute__((section (".logger"))) = DBG_FILE; \
00033     const static volatile uint16_t cookie __attribute__((section (".logger"))) = 0xCAFE; \
00034 }
```

15.49 /home/vm/mark3/trunk/embedded/kernel/public/driver.h File Reference

Driver abstraction framework.

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

Classes

- class Driver

  Base device-driver class used in hardware abstraction.
- class DriverList

  List of Driver objects used to keep track of all device drivers in the system.

15.49.1 Detailed Description

Driver abstraction framework.

15.49.2 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.

15.49.3 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.

15.49.4 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::FindBy↩ Name("/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.

15.50 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 uint8_t Open() = 0;
00139
00147     virtual uint8_t Close() = 0;
00148
00164     virtual uint16_t Read( uint16_t u16Bytes_,
00165                                uint8_t *pu8Data_) = 0;
00166
00183     virtual uint16_t Write( uint16_t u16Bytes_,
00184                                uint8_t *pu8Data_) = 0;
00185
00208     virtual uint16_t Control( uint16_t u16Event_,
00209                                 void *pvDataIn_,
00210                                 uint16_t u16SizeIn_,
00211                                 void *pvDataOut_,
00212                                 uint16_t u16SizeOut_ ) = 0;
00213
00222     void SetName( const char *pcName_ ) { m_pcPath = pcName_; }
00223
00231     const char *GetPath() { return m_pcPath; }
00232
00233 private:
00234
00236     const 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 char *m_pcPath );
00283
00284 private:
00285
00287     static DoubleLinkList m_clDriverList;
00288 };
00289
00290 #endif //KERNEL_USE_DRIVER
00291
00292 #endif
```

15.51 /home/vm/mark3/trunk/embedded/kernel/public/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.

15.51.1 Detailed Description

Event Flag Blocking Object/IPC-Object definition.

Definition in file eventflag.h.

15.52 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_u16SetMask = 0; m_clBlockList.
    Init(); }
00053
00061     uint16_t Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_);
00062
00063 #if KERNEL_USE_TIMEOUTS
00064
00072     uint16_t Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_, uint32_t
    u32TimeMS_);
00073
00081     void WakeMe(Thread *pclOwner_);
00082
00083 #endif
00084
00090     void Set(uint16_t u16Mask_);
00091
00096     void Clear(uint16_t u16Mask_);
00097
00102     uint16_t GetMask();
00103
00104 private:
00105
00106 #if KERNEL_USE_TIMEOUTS
00107
00119     uint16_t Wait_i(uint16_t u16Mask_, EventFlagOperation_t eMode_, uint32_t
    u32TimeMS_);
00120 #else
00121
00131     uint16_t Wait_i(uint16_t u16Mask_, EventFlagOperation_t eMode_);
00132 #endif
00133
00134     uint16_t m_u16SetMask;
00135 };
00136
00137 #endif //KERNEL_USE_EVENTFLAG
00138 #endif //__EVENTFLAG_H__
00139
```

15.53 /home/vm/mark3/trunk/embedded/kernel/public/kernel.h File Reference

Kernel initialization and startup class.

```
#include "mark3cfg.h"
#include "kerneltypes.h"
#include "paniccodes.h"
#include "thread.h"
```

Classes

- class Kernel

  Class that encapsulates all of the kernel startup functions.

15.53.1 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.

15.54 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 "mark3cfg.h"
00036 #include "kerneltypes.h"
00037 #include "paniccodes.h"
00038 #include "thread.h"
00039
00040 #if KERNEL_USE_IDLE_FUNC
00041 typedef void (*idle_func_t)(void);
00042 #endif
00043
00044 //---------------------------------------------------------------------------
00048 class Kernel
00049 {
00050 public:
00059     static void Init(void);
00060
00073     static void Start(void);
00074
00080     static bool IsStarted()    {   return m_bIsStarted;    }
00081
00089     static void SetPanic( panic_func_t pfPanic_ ) {
      m_pfPanic = pfPanic_; }
00090
00095     static bool IsPanic()      {   return m_bIsPanic;   }
00096
00101     static void Panic(uint16_t u16Cause_);
00102
00103 #if KERNEL_USE_IDLE_FUNC
00104
00109     static void SetIdleFunc( idle_func_t pfIdle_ )  {   m_pfIdle = pfIdle_; }
```

```
00110
00115     static void IdleFunc(void) { if (m_pfIdle != 0 ){ m_pfIdle(); } }
00116
00124     static Thread *GetIdleThread(void) { return (Thread*)&
      m_clIdle; }
00125 #endif
00126
00127 private:
00128     static bool m_bIsStarted;
00129     static bool m_bIsPanic;
00130     static panic_func_t m_pfPanic;
00131 #if KERNEL_USE_IDLE_FUNC
00132     static idle_func_t m_pfIdle;
00133     static FakeThread_t m_clIdle;
00134 #endif
00135 };
00136
00137 #endif
00138
```

15.55 /home/vm/mark3/trunk/embedded/kernel/public/kernelaware.h File Reference

Kernel aware simulation support.

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

Classes

- class KernelAware

  The KernelAware class.

Enumerations

- enum KernelAwareCommand_t {
  KA_COMMAND_IDLE = 0, KA_COMMAND_PROFILE_INIT, KA_COMMAND_PROFILE_START, KA_CO↩
  MMAND_PROFILE_STOP,
  KA_COMMAND_PROFILE_REPORT, KA_COMMAND_EXIT_SIMULATOR, KA_COMMAND_TRACE_0,
  KA_COMMAND_TRACE_1,
  KA_COMMAND_TRACE_2, KA_COMMAND_PRINT }

  This enumeration contains a list of supported commands that can be executed to invoke a response from a kernel aware host.

15.55.1 Detailed Description

Kernel aware simulation support.

Definition in file kernelaware.h.

15.55.2 Enumeration Type Documentation

15.55.2.1 enum KernelAwareCommand_t

This enumeration contains a list of supported commands that can be executed to invoke a response from a kernel aware host.

Enumerator

KA_COMMAND_IDLE Null command, does nothing.

KA_COMMAND_PROFILE_INIT Initialize a new profiling session.

KA_COMMAND_PROFILE_START Begin a profiling sample.

KA_COMMAND_PROFILE_STOP End a profiling sample.

KA_COMMAND_PROFILE_REPORT Report current profiling session.

KA_COMMAND_EXIT_SIMULATOR Terminate the host simulator.

KA_COMMAND_TRACE_0 0-argument kernel trace

KA_COMMAND_TRACE_1 1-argument kernel trace

KA_COMMAND_TRACE_2 2-argument kernel trace

KA_COMMAND_PRINT Print an arbitrary string of data.

Definition at line 33 of file kernelaware.h.

15.56 kernelaware.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 __KERNEL_AWARE_H__
00022 #define __KERNEL_AWARE_H__
00023
00024 #include "kerneltypes.h"
00025 #include "mark3cfg.h"
00026
00027 #if KERNEL_AWARE_SIMULATION
00028 //---------------------------------------------------------------------------
00033 typedef enum
00034 {
00035     KA_COMMAND_IDLE = 0,
00036     KA_COMMAND_PROFILE_INIT,
00037     KA_COMMAND_PROFILE_START,
00038     KA_COMMAND_PROFILE_STOP,
00039     KA_COMMAND_PROFILE_REPORT,
00040     KA_COMMAND_EXIT_SIMULATOR,
00041     KA_COMMAND_TRACE_0,
00042     KA_COMMAND_TRACE_1,
00043     KA_COMMAND_TRACE_2,
00044     KA_COMMAND_PRINT
00045 } KernelAwareCommand_t;
00046
00047 //---------------------------------------------------------------------------
00065 class KernelAware
00066 {
00067 public:
00068     //-----------------------------------------------------------------------
00079     static void ProfileInit( const char *szStr_ );
00080
00081     //-----------------------------------------------------------------------
00089     static void ProfileStart( void );
00090
00091     //-----------------------------------------------------------------------
00098     static void ProfileStop( void );
00099
00100     //-----------------------------------------------------------------------
00108     static void ProfileReport( void );
00109
00110     //-----------------------------------------------------------------------
00118     static void ExitSimulator( void );
00119
00120     //-----------------------------------------------------------------------
00128     static void Print( const char *szStr_ );
00129
00130     //-----------------------------------------------------------------------
00140     static void Trace( uint16_t u16File_,
00141                        uint16_t u16Line_);
```

```
00142
00143     //---------------------------------------------------------------------------
00154     static void Trace( uint16_t u16File_,
00155                        uint16_t u16Line_,
00156                        uint16_t u16Arg1_);
00157
00158     //---------------------------------------------------------------------------
00170     static void Trace( uint16_t u16File_,
00171                        uint16_t u16Line_,
00172                        uint16_t u16Arg1_,
00173                        uint16_t u16Arg2_);
00174
00175     //---------------------------------------------------------------------------
00185     static bool IsSimulatorAware(void);
00186
00187 private:
00188
00189    //---------------------------------------------------------------------------
00202     static void Trace_i( uint16_t u16File_,
00203                          uint16_t u16Line_,
00204                          uint16_t u16Arg1_,
00205                          uint16_t u16Arg2_,
00206                          KernelAwareCommand_t eCmd_);
00207 };
00208
00209 #endif
00210
00211 #endif
```

15.57 /home/vm/mark3/trunk/embedded/kernel/public/kerneldebug.h File Reference

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

```
#include "mark3cfg.h"
#include "tracebuffer.h"
#include "kernelaware.h"
#include "paniccodes.h"
#include "kernel.h"
#include "buffalogger.h"
#include "dbg_file_list.h"
```

15.57.1 Detailed Description

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

Definition in file kerneldebug.h.

15.58 kerneldebug.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 "mark3cfg.h"
00024 #include "tracebuffer.h"
00025 #include "kernelaware.h"
00026 #include "paniccodes.h"
00027 #include "kernel.h"
```

```
00028 #include "buffalogger.h"
00029 #include "dbg_file_list.h"
00030
00031 //---------------------------------------------------------------------------
00032 #if (KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION)
00033
00034 //---------------------------------------------------------------------------
00035 #define KERNEL_TRACE( x )   \
00036 {   \
00037     EMIT_DBG_STRING( x ); \
00038     uint16_t au16Msg__[5]; \
00039     au16Msg__[0] = 0xACDC;  \
00040     au16Msg__[1] = DBG_FILE; \
00041     au16Msg__[2] = __LINE__; \
00042     au16Msg__[3] = TraceBuffer::Increment() ; \
00043     TraceBuffer::Write(au16Msg__, 5); \
00044 };
00045
00046 //---------------------------------------------------------------------------
00047 #define KERNEL_TRACE_1( x, arg1 ) \
00048 {   \
00049     EMIT_DBG_STRING( x ); \
00050     uint16_t au16Msg__[6]; \
00051     au16Msg__[0] = 0xACDC;  \
00052     au16Msg__[1] = DBG_FILE; \
00053     au16Msg__[2] = __LINE__; \
00054     au16Msg__[3] = TraceBuffer::Increment(); \
00055     au16Msg__[4] = arg1; \
00056     TraceBuffer::Write(au16Msg__, 6); \
00057 }
00058
00059 //---------------------------------------------------------------------------
00060 #define KERNEL_TRACE_2( x, arg1, arg2 ) \
00061 {   \
00062     EMIT_DBG_STRING( x ); \
00063     uint16_t au16Msg__[7]; \
00064     au16Msg__[0] = 0xACDC;  \
00065     au16Msg__[1] = DBG_FILE; \
00066     au16Msg__[2] = __LINE__; \
00067     au16Msg__[3] = TraceBuffer::Increment(); \
00068     au16Msg__[4] = arg1; \
00069     au16Msg__[5] = arg2; \
00070     TraceBuffer::Write(au16Msg__, 7); \
00071 }
00072
00073 //---------------------------------------------------------------------------
00074 #define KERNEL_ASSERT( x ) \
00075 {       \
00076     if( ( x ) == false ) \
00077     {   \
00078         EMIT_DBG_STRING( x ); \
00079         uint16_t au16Msg__[5];  \
00080         au16Msg__[0] = 0xACDC;  \
00081         au16Msg__[1] = DBG_FILE;    \
00082         au16Msg__[2] = __LINE__; \
00083         au16Msg__[3] = TraceBuffer::Increment(); \
00084         TraceBuffer::Write(au16Msg__, 5); \
00085         Kernel::Panic(PANIC_ASSERT_FAILED); \
00086     }   \
00087 }
00088
00089 #elif (KERNEL_USE_DEBUG && KERNEL_AWARE_SIMULATION)
00090
00091 //---------------------------------------------------------------------------
00092 #define KERNEL_TRACE( x )   \
00093 {   \
00094     EMIT_DBG_STRING( x ); \
00095     KernelAware::Trace( DBG_FILE, __LINE__ ); \
00096 };
00097
00098 //---------------------------------------------------------------------------
00099 #define KERNEL_TRACE_1( x, arg1 ) \
00100 {   \
00101     EMIT_DBG_STRING( x ); \
00102     KernelAware::Trace( DBG_FILE, __LINE__, arg1 ); \
00103 }
00104
00105 //---------------------------------------------------------------------------
00106 #define KERNEL_TRACE_2( x, arg1, arg2 ) \
00107 {   \
00108     EMIT_DBG_STRING( x ); \
00109     KernelAware::Trace( DBG_FILE, __LINE__, arg1, arg2 ); \
00110 }
00111
00112 //---------------------------------------------------------------------------
00113 #define KERNEL_ASSERT( x ) \
00114 {       \
```

```
00115     if( ( x ) == false ) \
00116     {   \
00117         EMIT_DBG_STRING( "ASSERT FAILED" ); \
00118         KernelAware::Trace( DBG_FILE, __LINE__ ); \
00119         Kernel::Panic( PANIC_ASSERT_FAILED ); \
00120     }   \
00121 }
00122
00123 #else
00124 //---------------------------------------------------------------------------
00125 // Note -- when kernel-debugging is disabled, we still have to define the
00126 // macros to ensure that the expressions compile (albeit, by elimination
00127 // during pre-processing).
00128 //---------------------------------------------------------------------------
00129 #define KERNEL_TRACE( x )
00130 //---------------------------------------------------------------------------
00131 #define KERNEL_TRACE_1( x, arg1 )
00132 //---------------------------------------------------------------------------
00133 #define KERNEL_TRACE_2( x, arg1, arg2 )
00134 //---------------------------------------------------------------------------
00135 #define KERNEL_ASSERT( x )
00136
00137 #endif // KERNEL_USE_DEBUG
00138
00139 #endif
```

15.59 /home/vm/mark3/trunk/embedded/kernel/public/kerneltypes.h File Reference

Basic data type primatives used throughout the OS.

`#include <stdint.h>`

Macros

- #define K_ADDR uint32_t

  Primative datatype representing address-size.

- #define K_WORD uint32_t

  Primative datatype representing a data word.

Typedefs

- typedef void(∗ panic_func_t)(uint16_t u16PanicCode_)

  Function pointer type used to implement kernel-panic handlers.

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 }

  This enumeration describes the different operations supported by the event flag blocking object.

15.59.1 Detailed Description

Basic data type primatives used throughout the OS.

Definition in file kerneltypes.h.

15.59.2 Enumeration Type Documentation

15.59.2.1 enum EventFlagOperation_t

This enumeration describes the different operations supported by the event flag blocking object.

Enumerator

EVENT_FLAG_ALL Block until all bits in the specified bitmask are set.

EVENT_FLAG_ANY Block until any bits in the specified bitmask are set.

EVENT_FLAG_ALL_CLEAR Block until all bits in the specified bitmask are cleared.

EVENT_FLAG_ANY_CLEAR Block until any bits in the specified bitmask are cleared.

EVENT_FLAG_MODES Count of event-flag modes. Not used by user

EVENT_FLAG_PENDING_UNBLOCK Special code. Not used by user

Definition at line 43 of file kerneltypes.h.

15.60 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 //---------------------------------------------------------------------------
00025 #if !defined(K_ADDR)
00026     #define K_ADDR      uint32_t
00027 #endif
00028 #if !defined(K_WORD)
00029     #define K_WORD      uint32_t
00030 #endif
00031
00032 //---------------------------------------------------------------------------
00036 typedef void (*panic_func_t)( uint16_t u16PanicCode_ );
00037
00038 //---------------------------------------------------------------------------
00043 typedef enum
00044 {
00045     EVENT_FLAG_ALL,
00046     EVENT_FLAG_ANY,
00047     EVENT_FLAG_ALL_CLEAR,
00048     EVENT_FLAG_ANY_CLEAR,
00049 //---
00050     EVENT_FLAG_MODES,
00051     EVENT_FLAG_PENDING_UNBLOCK
00052 } EventFlagOperation_t;
00053
00054 #endif
```

15.61 /home/vm/mark3/trunk/embedded/kernel/public/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.

15.61.1 Detailed Description

Semaphore Blocking Object class declarations.

Definition in file ksemaphore.h.

15.62 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(uint16_t u16InitVal_, uint16_t u16MaxVal_);
00050
00059     bool Post();
00060
00067     void Pend();
00068
00080     uint16_t GetCount();
00081
00082 #if KERNEL_USE_TIMEOUTS
00083
00094     bool Pend( uint32_t u32WaitTimeMS_);
00095
00106     void WakeMe(Thread *pclChosenOne_);
00107 #endif
00108
00109 private:
00110
00116     uint8_t WakeNext();
00117
00118 #if KERNEL_USE_TIMEOUTS
00119
00127     bool Pend_i( uint32_t u32WaitTimeMS_ );
00128 #else
00129
00135     void Pend_i( void );
00136 #endif
```

```
00137
00138     uint16_t m_u16Value;
00139     uint16_t m_u16MaxValue;
00140
00141
00142 };
00143
00144 #endif //KERNEL_USE_SEMAPHORE
00145
00146 #endif
```

15.63 /home/vm/mark3/trunk/embedded/kernel/public/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.

15.63.1 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 u16 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.

15.64 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
```

15.65 /home/vm/mark3/trunk/embedded/kernel/public/mailbox.h File Reference

Mailbox + Envelope IPC Mechanism.

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

15.65.1 Detailed Description

Mailbox + Envelope IPC Mechanism.

Definition in file mailbox.h.

15.66 mailbox.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 __MAILBOX_H__
00022 #define __MAILBOX_H__
00023
00024 #include "mark3cfg.h"
00025 #include "kerneltypes.h"
00026 #include "ksemaphore.h"
00027
00028 #if KERNEL_USE_MAILBOX
00029
00030 class Mailbox
00031 {
00032 public:
00033
00044     void Init( void *pvBuffer_, uint16_t u16BufferSize_, uint16_t uselementSize_ );
00045
00059     bool Send( void *pvData_ );
00060
00074     bool SendTail( void *pvData_ );
00075
00076 #if KERNEL_USE_TIMEOUTS
00077
00091     bool Send( void *pvData_, uint32_t u32TimeoutMS_ );
00092
00107     bool SendTail( void *pvData_, uint32_t u32TimeoutMS_ );
00108  #endif
00109
00119     void Receive( void *pvData_ );
00120
00130     void ReceiveTail( void *pvData_ );
00131
00132 #if KERNEL_USE_TIMEOUTS
00133
00145     bool Receive( void *pvData_, uint32_t u32TimeoutMS_ );
00146
00159     bool ReceiveTail( void *pvData_, uint32_t u32TimeoutMS_ );
00160 #endif
00161
00162     uint16_t GetFreeSlots( void )
00163     {
00164         uint16_t rc;
00165         CS_ENTER();
00166         rc = m_u16Free;
00167         CS_EXIT();
00168         return rc;
00169     }
00170
```

```
00171     bool IsFull( void )
00172     {
00173         return (GetFreeSlots() == 0);
00174     }
00175 
00176     bool IsEmpty( void )
00177     {
00178         return (GetFreeSlots() == m_u16Count);
00179     }
00180 
00181 private:
00182 
00191     void *GetHeadPointer(void)
00192     {
00193         K_ADDR uAddr = (K_ADDR)m_pvBuffer;
00194         uAddr += (K_ADDR)(m_uselementSize) * (K_ADDR)(m_u16Head);
00195         return (void*)uAddr;
00196     }
00197 
00206     void *GetTailPointer(void)
00207     {
00208         K_ADDR uAddr = (K_ADDR)m_pvBuffer;
00209         uAddr += (K_ADDR)(m_uselementSize) * (K_ADDR)(m_u16Tail);
00210         return (void*)uAddr;
00211     }
00212 
00222     void CopyData( const void *src_, const void *dst_, uint16_t len_ )
00223     {
00224         uint8_t *u8Src = (uint8_t*)src_;
00225         uint8_t *u8Dst = (uint8_t*)dst_;
00226         while (len_--)
00227         {
00228             *u8Dst++ = *u8Src++;
00229         }
00230     }
00231 
00237     void MoveTailForward(void)
00238     {
00239         m_u16Tail++;
00240         if (m_u16Tail == m_u16Count)
00241         {
00242             m_u16Tail = 0;
00243         }
00244     }
00245 
00251     void MoveHeadForward(void)
00252     {
00253         m_u16Head++;
00254         if (m_u16Head == m_u16Count)
00255         {
00256             m_u16Head = 0;
00257         }
00258     }
00259 
00265     void MoveTailBackward(void)
00266     {
00267         if (m_u16Tail == 0)
00268         {
00269             m_u16Tail = m_u16Count;
00270         }
00271         m_u16Tail--;
00272     }
00273 
00279     void MoveHeadBackward(void)
00280     {
00281         if (m_u16Head == 0)
00282         {
00283             m_u16Head = m_u16Count;
00284         }
00285         m_u16Head--;
00286     }
00287 
00288 #if KERNEL_USE_TIMEOUTS
00289 
00299     bool Send_i( const void *pvData_, bool bTail_, uint32_t u32WaitTimeMS_ );
00300 #else
00301 
00310     bool Send_i( const void *pvData_, bool bTail_ );
00311 #endif
00312 
00313 #if KERNEL_USE_TIMEOUTS
00314 
00324     bool Receive_i( const void *pvData_, bool bTail_, uint32_t u32WaitTimeMS_ );
00325 #else
00326 
00334     void Receive_i( const void *pvData_, bool bTail_ );
00335 #endif
```

```
00336
00337     uint16_t m_u16Head;
00338     uint16_t m_u16Tail;
00339
00340     uint16_t m_u16Count;
00341     volatile uint16_t m_u16Free;
00342
00343     uint16_t m_uselementSize;
00344     const void *m_pvBuffer;
00345
00346     Semaphore m_clRecvSem;
00347
00348 #if KERNEL_USE_TIMEOUTS
00349     Semaphore m_clSendSem;
00350 #endif
00351
00352 };
00353
00354 #endif
00355
00356 #endif
00357
```

15.67 /home/vm/mark3/trunk/embedded/kernel/public/manual.h File Reference

Ascii-format documentation, used by doxygen to create various printable and viewable forms.

15.67.1 Detailed Description

Ascii-format documentation, used by doxygen to create various printable and viewable forms.

Definition in file manual.h.

15.68 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 ===========================================================================*/
```

15.69 /home/vm/mark3/trunk/embedded/kernel/public/mark3.h File Reference

Single include file given to users of the Mark3 Kernel API.

```
#include "mark3cfg.h"
#include "kerneltypes.h"
#include "threadport.h"
#include "kernelswi.h"
#include "kerneltimer.h"
#include "kernelprofile.h"
#include "kernel.h"
#include "thread.h"
#include "timerlist.h"
#include "ksemaphore.h"
#include "mutex.h"
#include "eventflag.h"
#include "message.h"
#include "notify.h"
#include "mailbox.h"
#include "atomic.h"
#include "driver.h"
#include "kernelaware.h"
#include "profile.h"
```

15.69.1 Detailed Description

Single include file given to users of the Mark3 Kernel API.

Definition in file mark3.h.

15.70 mark3.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 __MARK3_H__
00022 #define __MARK3_H__
00023
00024 #include "mark3cfg.h"
00025 #include "kerneltypes.h"
00026
00027 #include "threadport.h"
00028 #include "kernelswi.h"
00029 #include "kerneltimer.h"
00030 #include "kernelprofile.h"
00031
00032 #include "kernel.h"
00033 #include "thread.h"
00034 #include "timerlist.h"
00035
00036 #include "ksemaphore.h"
00037 #include "mutex.h"
00038 #include "eventflag.h"
00039 #include "message.h"
00040 #include "notify.h"
00041 #include "mailbox.h"
00042
00043 #include "atomic.h"
00044 #include "driver.h"
00045
00046 #include "kernelaware.h"
00047
00048 #include "profile.h"
```

```
00049 #endif
```

15.71 /home/vm/mark3/trunk/embedded/kernel/public/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_TIMERS_TICKLESS (1)

  If you've opted to use the kernel timers module, you have an option as to which timer implementation to use: Tick-based or Tick-less.
- #define KERNEL_USE_TIMEOUTS (1)

  By default, if you opt to enable kernel timers, you also get timeout- enabled versions of the blocking object APIs along with it.
- #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 THREAD_QUANTUM_DEFAULT (4)

  This value defines the default thread quantum when KERNEL_USE_QUANTUM is enabled.
- #define KERNEL_USE_NOTIFY (1)

  This is a simple blocking object, where a thread (or threads) are guaranteed to block until an asynchronous event signals the object.
- #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_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_EVENTFLAG (1)

  Provides additional event-flag based blocking.
- #define KERNEL_USE_MESSAGE (1)

  Enable inter-thread messaging using message queues.
- #define GLOBAL_MESSAGE_POOL_SIZE (8)

  If Messages are enabled, define the size of the default kernel message pool.
- #define KERNEL_USE_MAILBOX (1)

  Enable inter-thread messaging using mailboxes.
- #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 (1)

  Enabling device drivers provides a posix-like filesystem interface for peripheral device drivers.
- #define KERNEL_USE_THREADNAME (0)

  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 (1)

  Provides extra classes for profiling the performance of code.
- #define KERNEL_USE_DEBUG (1)

  Provides extra logic for kernel debugging, and instruments the kernel with extra asserts, and kernel trace functionality.
- #define KERNEL_USE_ATOMIC (0)

Provides support for atomic operations, including addition, subtraction, set, and test-and-set.

- #define SAFE_UNLINK (0)

  "Safe unlinking" performs extra checks on data to make sure that there are no consistencies when performing operations on linked lists.

- #define KERNEL_AWARE_SIMULATION (1)

  Include support for kernel-aware simulation.

- #define KERNEL_USE_IDLE_FUNC (1)

  Enabling this feature removes the necessity for the user to dedicate a complete thread for idle functionality.

15.71.1 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.

Definition in file mark3cfg.h.

15.71.2 Macro Definition Documentation

15.71.2.1 #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. All message queues share their message objects from this global pool to maximize efficiency and simplify data management.

Definition at line 150 of file mark3cfg.h.

15.71.2.2 #define KERNEL_AWARE_SIMULATION (1)

Include support for kernel-aware simulation.

Enabling this feature adds advanced profiling, trace, and environment-aware debugging and diagnostic functionality when Mark3-based applications are run on the flavr AVR simulator.

Definition at line 231 of file mark3cfg.h.

15.71.2.3 #define KERNEL_TIMERS_TICKLESS (1)

If you've opted to use the kernel timers module, you have an option as to which timer implementation to use: Tick-based or Tick-less.

Tick-based timers provide a "traditional" RTOS timer implementation based on a fixed-frequency timer interrupt. While this provides very accurate, reliable timing, it also means that the CPU is being interrupted far more often than may be necessary (as not all timer ticks result in "real work" being done).

Tick-less timers still rely on a hardware timer interrupt, but uses a dynamic expiry interval to ensure that the interrupt is only called when the next timer expires. This increases the complexity of the timer interrupt handler, but reduces the number and frequency.

Note that the CPU port (kerneltimer.cpp) must be implemented for the particular timer variant desired.

Definition at line 62 of file mark3cfg.h.

15.71.2.4 #define KERNEL_USE_ATOMIC (0)

Provides support for atomic operations, including addition, subtraction, set, and test-and-set.

Add/Sub/Set contain 8, 16, and 32-bit variants.

Definition at line 215 of file mark3cfg.h.

15.71.2.5 #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 197 of file mark3cfg.h.

15.71.2.6 #define KERNEL_USE_EVENTFLAG (1)

Provides additional event-flag based blocking.

This relies on an additional per-thread flag-mask to be allocated, which adds 2 bytes to the size of each thread object.

Definition at line 129 of file mark3cfg.h.

15.71.2.7 #define KERNEL_USE_IDLE_FUNC (1)

Enabling this feature removes the necessity for the user to dedicate a complete thread for idle functionality.

This saves a full thread stack, but also requires a bit extra static data. This also adds a slight overhead to the context switch and scheduler, as a special case has to be taken into account.

Definition at line 240 of file mark3cfg.h.

15.71.2.8 #define KERNEL_USE_MAILBOX (1)

Enable inter-thread messaging using mailboxes.

A mailbox manages a blob of data provided by the user, that is partitioned into fixed-size blocks called envelopes. The size of an envelope is set by the user when the mailbox is initialized. Any number of threads can read-from and write-to the mailbox. Envelopes can be sent-to or received-from the mailbox at the head or tail. In this way, mailboxes essentially act as a circular buffer that can be used as a blocking FIFO or LIFO queue.

Definition at line 163 of file mark3cfg.h.

15.71.2.9 #define KERNEL_USE_MESSAGE (1)

Enable inter-thread messaging using message queues.

This is the preferred mechanism for IPC for serious multi-threaded communications; generally anywhere a semaphore or event-flag is insufficient.

Definition at line 137 of file mark3cfg.h.

15.71.2.10 #define KERNEL_USE_PROFILER (1)

Provides extra classes for profiling the performance of code.

useful for debugging and development, but uses an additional hardware timer.

Definition at line 203 of file mark3cfg.h.

15.71.2.11 #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 92 of file mark3cfg.h.

15.71.2.12 #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.

Definition at line 115 of file mark3cfg.h.

15.71.2.13 #define KERNEL_USE_THREADNAME (0)

Provide Thread method to allow the user to set a name for each thread in the system.

Adds a const char* pointer to the size of the thread object.

Definition at line 189 of file mark3cfg.h.

15.71.2.14 #define KERNEL_USE_TIMEOUTS (1)

By default, if you opt to enable kernel timers, you also get timeout- enabled versions of the blocking object APIs along with it.

This support comes at a small cost to code size, but a slightly larger cost to realtime performance - as checking for the use of timers in the underlying internal code costs some cycles.

As a result, the option is given to the user here to manually disable these timeout-based APIs if desired by the user for performance and code-size reasons.

Definition at line 77 of file mark3cfg.h.

15.71.2.15 #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 (usedd for round-robin scheduling) is dependent on this module, as is Thread Sleep functionality.

Definition at line 41 of file mark3cfg.h.

15.71.2.16 #define SAFE_UNLINK (0)

"Safe unlinking" performs extra checks on data to make sure that there are no consistencies when performing operations on linked lists.

This goes beyond pointer checks, adding a layer of structural and metadata validation to help detect system corruption early.

Definition at line 223 of file mark3cfg.h.

15.71.2.17 #define THREAD_QUANTUM_DEFAULT (4)

This value defines the default thread quantum when KERNEL_USE_QUANTUM is enabled.

The thread quantum value is in milliseconds

Definition at line 101 of file mark3cfg.h.

15.72 mark3cfg.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 ===========================================================================*/
00029 #ifndef __MARK3CFG_H__
00030 #define __MARK3CFG_H__
00031
00041 #define KERNEL_USE_TIMERS               (1)
00042
00061 #if KERNEL_USE_TIMERS
00062     #define KERNEL_TIMERS_TICKLESS      (1)
00063 #endif
00064
00076 #if KERNEL_USE_TIMERS
00077     #define KERNEL_USE_TIMEOUTS         (1)
00078 #else
00079     #define KERNEL_USE_TIMEOUTS         (0)
00080 #endif
00081
00091 #if KERNEL_USE_TIMERS
00092     #define KERNEL_USE_QUANTUM          (1)
00093 #else
00094     #define KERNEL_USE_QUANTUM          (0)
00095 #endif
00096
00101 #define THREAD_QUANTUM_DEFAULT          (4)
00102
00107 #define KERNEL_USE_NOTIFY               (1)
00108
00115 #define KERNEL_USE_SEMAPHORE            (1)
00116
00122 #define KERNEL_USE_MUTEX                (1)
00123
00129 #define KERNEL_USE_EVENTFLAG            (1)
00130
00136 #if KERNEL_USE_SEMAPHORE
00137     #define KERNEL_USE_MESSAGE          (1)
00138 #else
00139     #define KERNEL_USE_MESSAGE          (0)
00140 #endif
00141
00149 #if KERNEL_USE_MESSAGE
00150     #define GLOBAL_MESSAGE_POOL_SIZE    (8)
00151 #endif
00152
00162 #if KERNEL_USE_SEMAPHORE
00163     #define KERNEL_USE_MAILBOX          (1)
00164 #else
00165     #define KERNEL_USE_MAILBOX          (0)
00166 #endif
00167
00172 #if KERNEL_USE_TIMERS && KERNEL_USE_SEMAPHORE
00173     #define KERNEL_USE_SLEEP            (1)
00174 #else
00175     #define KERNEL_USE_SLEEP            (0)
00176 #endif
00177
00182 #define KERNEL_USE_DRIVER               (1)
00183
00189 #define KERNEL_USE_THREADNAME           (0)
00190
00197 #define KERNEL_USE_DYNAMIC_THREADS      (1)
00198
```

```
00203 #define KERNEL_USE_PROFILER             (1)
00204
00209 #define KERNEL_USE_DEBUG                (1)
00210
00215 #define KERNEL_USE_ATOMIC               (0)
00216
00223 #define SAFE_UNLINK                     (0)
00224
00231 #define KERNEL_AWARE_SIMULATION         (1)
00232
00240 #define KERNEL_USE_IDLE_FUNC            (1)
00241 #endif
```

15.73 /home/vm/mark3/trunk/embedded/kernel/public/message.h File Reference

Inter-thread communication via message-passing.

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

15.73.1 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.

15.73.2 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.

15.74 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_u16Code = 0; }
00108
00116     void SetData( void *pvData_ ) { m_pvData = pvData_; }
00117
00125     void *GetData() { return m_pvData; }
00126
00134     void SetCode( uint16_t u16Code_ ) { m_u16Code = u16Code_; }
00135
00143     uint16_t GetCode() { return m_u16Code; }
00144 private:
00145
00147     void *m_pvData;
00148
00150     uint16_t m_u16Code;
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( uint32_t u32TimeWaitMS_ );
00237 #endif
00238
00247     void Send( Message *pclSrc_ );
00248
00249
00257     uint16_t GetCount();
00258 private:
00259
00260 #if KERNEL_USE_TIMEOUTS
00261
00270     Message *Receive_i( uint32_t u32TimeWaitMS_ );
00271 #else
00272
00279     Message *Receive_i( void );
00280 #endif
00281
00283     Semaphore m_clSemaphore;
00284
00286     DoubleLinkList m_clLinkList;
00287 };
00288
00289 #endif //KERNEL_USE_MESSAGE
00290
00291 #endif
```

15.75 /home/vm/mark3/trunk/embedded/kernel/public/mutex.h File Reference

Mutual exclusion class declaration.

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

Classes

- class Mutex

  Mutual-exclusion locks, based on BlockingObject.

15.75.1 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.

15.75.2 Initializing

Initializing a mutex object by calling:

```
clMutex.Init();
```

15.75.3 Resource protection example

```
clMutex.Claim();
...
<resource protected block>
...
clMutex.Release();
```

Definition in file mutex.h.

15.76 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(uint32_t u32WaitTimeMS_);
00098
00111     void WakeMe( Thread *pclOwner_ );
00112
00113 #endif
00114
00121     void Release();
00122
00123 private:
00124
00130     uint8_t WakeNext();
00131
00132
00133 #if KERNEL_USE_TIMEOUTS
00134
```

```
00142     bool Claim_i( uint32_t u32WaitTimeMS_ );
00143 #else
00144
00150     void Claim_i(void);
00151 #endif
00152
00153     uint8_t m_u8Recurse;
00154     bool m_bReady;
00155     uint8_t m_u8MaxPri;
00156     Thread *m_pclOwner;
00157
00158 };
00159
00160 #endif //KERNEL_USE_MUTEX
00161
00162 #endif //__MUTEX_H_
00163
```

15.77 /home/vm/mark3/trunk/embedded/kernel/public/notify.h File Reference

Lightweight thread notification - blocking object.

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

15.77.1 Detailed Description

Lightweight thread notification - blocking object.

Definition in file notify.h.

15.78 notify.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 __NOTIFY_H__
00023 #define __NOTIFY_H__
00024
00025 #include "mark3cfg.h"
00026 #include "blocking.h"
00027
00028 #if KERNEL_USE_NOTIFY
00029
00030 class Notify : public BlockingObject
00031 {
00032 public:
00038     void Init(void);
00039
00049     void Signal(void);
00050
00060     void Wait( bool *pbFlag_ );
00061
00062 #if KERNEL_USE_TIMEOUTS
00063
00074     bool Wait( uint32_t u32WaitTimeMS_, bool *pbFlag_ );
00075 #endif
00076
00086     void WakeMe(Thread *pclChosenOne_);
00087 };
00088
00089 #endif
00090
```

```
00091 #endif
```

15.79 /home/vm/mark3/trunk/embedded/kernel/public/paniccodes.h File Reference

Defines the reason codes thrown when a kernel panic occurs.

15.79.1 Detailed Description

Defines the reason codes thrown when a kernel panic occurs.

Definition in file paniccodes.h.

15.80 paniccodes.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
```

15.81 /home/vm/mark3/trunk/embedded/kernel/public/profile.h File Reference

High-precision profiling timers.

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

Classes

- class ProfileTimer

  Profiling timer.

15.81.1 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
u32AverageTimer = clMyTimer.GetAverage();

// Get the execution time from the last iteration
u32LastTimer = clMyTimer.GetCurrent();
```

Definition in file profile.h.

15.82 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     uint32_t GetAverage();
00105
00114     uint32_t GetCurrent();
00115
00116 private:
00117
00126     uint32_t ComputeCurrentTicks(uint16_t u16Count_, uint32_t u32Epoch_);
00127
00128     uint32_t m_u32Cumulative;
00129     uint32_t m_u32CurrentIteration;
00130     uint16_t m_u16Initial;
00131     uint32_t m_u32InitialEpoch;
00132     uint16_t m_u16Iterations;
00133     bool m_bActive;
00134 };
00135
00136 #endif // KERNEL_USE_PROFILE
00137
00138 #endif
```

15.83 /home/vm/mark3/trunk/embedded/kernel/public/quantum.h File Reference

Thread Quantum declarations for Round-Robin Scheduling.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "thread.h"
#include "timer.h"
#include "timerlist.h"
#include "timerscheduler.h"
```

Classes

- class Quantum

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

15.83.1 Detailed Description

Thread Quantum declarations for Round-Robin Scheduling.

Definition in file quantum.h.

15.84 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 "timer.h"
00030 #include "timerlist.h"
00031 #include "timerscheduler.h"
00032
00033 #if KERNEL_USE_QUANTUM
00034 class Timer;
00035
00041 class Quantum
00042 {
00043 public:
00052     static void UpdateTimer();
00053
00060     static void AddThread( Thread *pclThread_ );
00061
00067     static void RemoveThread();
00068
00077     static void SetInTimer(void)   { m_bInTimer = true; }
00078
00084     static void ClearInTimer(void) { m_bInTimer = false; }
00085
00086 private:
00098     static void SetTimer( Thread *pclThread_ );
00099
00100     static Timer m_clQuantumTimer;
00101     static bool  m_bActive;
00102     static bool  m_bInTimer;
```

```
00103 };
00104
00105 #endif //KERNEL_USE_QUANTUM
00106
00107 #endif
```

15.85 /home/vm/mark3/trunk/embedded/kernel/public/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)

  Defines the maximum number of thread priorities supported in the scheduler.

Variables

- volatile Thread * g_pclNext

  Pointer to the currently-chosen next-running thread.
- Thread * g_pclCurrent

  Pointer to the currently-running thread.

15.85.1 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.

15.86 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 volatile Thread *g_pclNext;
00054 extern Thread *g_pclCurrent;
00055
00056 #define NUM_PRIORITIES              (8)
00057 //---------------------------------------------------------------------------
00058
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 bool SetScheduler(bool bEnable_);
00113
00119     static Thread *GetCurrentThread(){ return g_pclCurrent; }
00120
00127     static volatile Thread *GetNextThread(){ return g_pclNext; }
00128
00137     static ThreadList *GetThreadList(uint8_t u8Priority_){ return &
      m_aclPriorities[u8Priority_]; }
00138
00145     static ThreadList *GetStopList(){ return &m_clStopList; }
00146
00155     static uint8_t IsEnabled(){ return m_bEnabled; }
00156
00163     static void QueueScheduler() { m_bQueuedSchedule = true; }
00164
00165 private:
00167     static bool m_bEnabled;
00168
00170     static bool m_bQueuedSchedule;
00171
00173     static ThreadList m_clStopList;
00174
00176     static ThreadList m_aclPriorities[NUM_PRIORITIES];
00177
00179     static uint8_t m_u8PriFlag;
00180 };
00181 #endif
00182
```

15.87 /home/vm/mark3/trunk/embedded/kernel/public/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.
- struct FakeThread_t

  If the kernel is set up to use an idle function instead of an idle thread, we use a placeholder data structure to "simulate" the effect of having an idle thread in the system.

Typedefs

- typedef void(∗ ThreadEntry_t)(void ∗pvArg_)

  Function pointer type used for thread entrypoint functions.

Enumerations

- enum ThreadState_t

  Enumeration representing the different states a thread can exist in.

15.87.1 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.

15.88 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 typedef enum
00058 {
00059     THREAD_STATE_EXIT = 0,
00060     THREAD_STATE_READY,
00061     THREAD_STATE_BLOCKED,
00062     THREAD_STATE_STOP,
00063 //--
00064     THREAD_STATES
00065 } ThreadState_t;
00066
00067 //---------------------------------------------------------------------------
00071 class Thread : public LinkListNode
00072 {
00073 public:
00093     void Init(K_WORD *paucStack_,
00094               uint16_t u16StackSize_,
00095               uint8_t u8Priority_,
00096               ThreadEntry_t pfEntryPoint_,
00097               void *pvArg_ );
00098
00106     void Start();
00107
00108
00115     void Stop();
00116
00117 #if KERNEL_USE_THREADNAME
00118
00127     void SetName(const char *szName_) { m_szName = szName_; }
00128
00135     const char* GetName() { return m_szName; }
00136 #endif
00137
00146     ThreadList *GetOwner(void) { return m_pclOwner; }
00147
00155     ThreadList *GetCurrent(void) { return m_pclCurrent; }
00156
00165     uint8_t GetPriority(void) { return m_u8Priority; }
00166
00174     uint8_t GetCurPriority(void) { return m_u8CurPriority; }
00175
00176 #if KERNEL_USE_QUANTUM
00177
00184     void SetQuantum( uint16_t u16Quantum_ ) { m_u16Quantum = u16Quantum_; }
00185
00193     uint16_t GetQuantum(void) { return m_u16Quantum; }
00194 #endif
00195
00203     void SetCurrent( ThreadList *pclNewList_ ) {
      m_pclCurrent = pclNewList_; }
00204
00212     void SetOwner( ThreadList *pclNewList_ ) { m_pclOwner = pclNewList_; }
00213
00214
00227     void SetPriority(uint8_t u8Priority_);
00228
00238     void InheritPriority(uint8_t u8Priority_);
00239
00240 #if KERNEL_USE_DYNAMIC_THREADS
00241
00252     void Exit();
00253 #endif
00254
00255 #if KERNEL_USE_SLEEP
00256
00264     static void Sleep(uint32_t u32TimeMs_);
00265
00274     static void USleep(uint32_t u32TimeUs_);
00275 #endif
00276
00284     static void Yield(void);
00285
00293     void SetID( uint8_t u8ID_ ) { m_u8ThreadID = u8ID_; }
```

```
00294
00302     uint8_t GetID() { return m_u8ThreadID; }
00303
00304
00317     uint16_t GetStackSlack();
00318
00319 #if KERNEL_USE_EVENTFLAG
00320
00327     uint16_t GetEventFlagMask() { return m_u16FlagMask; }
00328
00333     void SetEventFlagMask(uint16_t u16Mask_) { m_u16FlagMask = u16Mask_; }
00334
00340     void SetEventFlagMode(EventFlagOperation_t eMode_ ) {
      m_eFlagMode = eMode_; }
00341
00346     EventFlagOperation_t GetEventFlagMode() { return
      m_eFlagMode; }
00347 #endif
00348
00349 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00350
00353     Timer *GetTimer();
00354 #endif
00355 #if KERNEL_USE_TIMEOUTS
00356
00364     void SetExpired( bool bExpired_ );
00365
00372     bool GetExpired();
00373 #endif
00374
00375 #if KERNEL_USE_IDLE_FUNC
00376
00381     void InitIdle();
00382 #endif
00383
00390     ThreadState_t GetState()                { return
      m_eState; }
00391
00399     void SetState( ThreadState_t eState_ )  { m_eState = eState_; }
00400
00401     friend class ThreadPort;
00402
00403 private:
00411     static void ContextSwitchSWI(void);
00412
00417     void SetPriorityBase(uint8_t u8Priority_);
00418
00420     K_WORD *m_pwStackTop;
00421
00423     K_WORD *m_pwStack;
00424
00426     uint8_t m_u8ThreadID;
00427
00429     uint8_t m_u8Priority;
00430
00432     uint8_t m_u8CurPriority;
00433
00435     ThreadState_t m_eState;
00436
00437 #if KERNEL_USE_THREADNAME
00438     const char *m_szName;
00440 #endif
00441
00443     uint16_t m_u16StackSize;
00444
00446     ThreadList *m_pclCurrent;
00447
00449     ThreadList *m_pclOwner;
00450
00452     ThreadEntry_t m_pfEntryPoint;
00453
00455     void *m_pvArg;
00456
00457 #if KERNEL_USE_QUANTUM
00458     uint16_t m_u16Quantum;
00460 #endif
00461
00462 #if KERNEL_USE_EVENTFLAG
00463     uint16_t m_u16FlagMask;
00465
00467     EventFlagOperation_t m_eFlagMode;
00468 #endif
00469
00470 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00471     Timer   m_clTimer;
00473 #endif
00474 #if KERNEL_USE_TIMEOUTS
```

```
00475     bool    m_bExpired;
00477 #endif
00478
00479 };
00480
00481 #if KERNEL_USE_IDLE_FUNC
00482 //---------------------------------------------------------------------------
00494 typedef struct
00495 {
00496     LinkListNode *next;
00497     LinkListNode *prev;
00498
00500     K_WORD *m_pwStackTop;
00501
00503     K_WORD *m_pwStack;
00504
00506     uint8_t m_u8ThreadID;
00507
00509     uint8_t m_u8Priority;
00510
00512     uint8_t m_u8CurPriority;
00513
00515     ThreadState_t m_eState;
00516
00517 #if KERNEL_USE_THREADNAME
00518     const char *m_szName;
00520 #endif
00521
00522 } FakeThread_t;
00523 #endif
00524
00525 #endif
```

15.89 /home/vm/mark3/trunk/embedded/kernel/public/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.

15.89.1 Detailed Description

Thread linked-list declarations.

Definition in file threadlist.h.

15.90 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_u8Priority = 0; m_pu8Flag = NULL; }
00041
00049     void SetPriority(uint8_t u8Priority_);
00050
00059     void SetFlagPointer(uint8_t *pu8Flag_);
00060
00068     void Add(LinkListNode *node_);
00069
00083     void Add(LinkListNode *node_, uint8_t *pu8Flag_, uint8_t u8Priority_);
00084
00092     void Remove(LinkListNode *node_);
00093
00101     Thread *HighestWaiter();
00102 private:
00103
00105     uint8_t m_u8Priority;
00106
00108     uint8_t *m_pu8Flag;
00109 };
00110
00111 #endif
00112
```

15.91 /home/vm/mark3/trunk/embedded/kernel/public/timer.h File Reference

Timer object 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.

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 MIN_TICKS (3)

  The minimum tick value to set.

Typedefs

- typedef void(∗ TimerCallback_t)(Thread ∗pclOwner_, void ∗pvData_)

  This type defines the callback function type for timer events.

15.91.1 Detailed Description

Timer object declarations.

Definition in file timer.h.

15.91.2 Macro Definition Documentation

15.91.2.1 #define TIMERLIST_FLAG_EXPIRED (0x08)

Timer is actually expired.

Definition at line 36 of file timer.h.

15.91.3 Typedef Documentation

15.91.3.1 typedef void(∗ TimerCallback_t)(Thread ∗pclOwner_, void ∗pvData_)

This type defines the callback function type for timer events.

Since these are called from an interrupt context, they do not operate from within a thread or object context directly – as a result, the context must be manually passed into the calls.

pclOwner_ is a pointer to the thread that owns the timer pvData_ is a pointer to some data or object that needs to know about the timer's expiry from within the timer interrupt context.

Definition at line 91 of file timer.h.

15.92 timer.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 __TIMER_H__
00022 #define __TIMER_H__
00023
00024 #include "kerneltypes.h"
00025 #include "mark3cfg.h"
00026
00027 #include "ll.h"
00028
00029 #if KERNEL_USE_TIMERS
00030 class Thread;
00031
00032 //---------------------------------------------------------------------------
00033 #define TIMERLIST_FLAG_ONE_SHOT         (0x01)
00034 #define TIMERLIST_FLAG_ACTIVE           (0x02)
00035 #define TIMERLIST_FLAG_CALLBACK         (0x04)
00036 #define TIMERLIST_FLAG_EXPIRED          (0x08)
00037
00038 //---------------------------------------------------------------------------
00039 #define MAX_TIMER_TICKS                 (0x7FFFFFFF)
```

```
00040
00041 //---------------------------------------------------------------------------
00042 #if KERNEL_TIMERS_TICKLESS
00043
00044 //---------------------------------------------------------------------------
00045 /*
00046     Ugly macros to support a wide resolution of delays.
00047     Given a 16-bit timer @ 16MHz & 256 cycle prescaler, this gives u16...
00048     Max time, SECONDS_TO_TICKS:  68719s
00049     Max time, MSECONDS_TO_TICKS: 6871.9s
00050     Max time, useCONDS_TO_TICKS: 6.8719s
00051
00052     ...With a 16us tick resolution.
00053
00054     Depending on the system frequency and timer resolution, you may want to
00055     customize these values to suit your system more appropriately.
00056  */
00057 //---------------------------------------------------------------------------
00058 #define SECONDS_TO_TICKS(x)             ((((uint32_t)x) * TIMER_FREQ))
00059 #define MSECONDS_TO_TICKS(x)            ((((((uint32_t)x) * (TIMER_FREQ/100)) + 5) / 10))
00060 #define useCONDS_TO_TICKS(x)            ((((((uint32_t)x) * TIMER_FREQ) + 50000) / 1000000))
00061
00062 //---------------------------------------------------------------------------
00063 #define MIN_TICKS                        (3)
00064 //---------------------------------------------------------------------------
00065
00066 #else
00067
00068 //---------------------------------------------------------------------------
00069 // add time because we don't know how far in an epoch we are when a call is made.
00070 #define SECONDS_TO_TICKS(x)             (((uint32_t)(x) * 1000) + 1)
00071 #define MSECONDS_TO_TICKS(x)            ((uint32_t)(x + 1))
00072 #define useCONDS_TO_TICKS(x)            (((uint32_t)(x + 999)) / 1000)
00073
00074 //---------------------------------------------------------------------------
00075 #define MIN_TICKS                       (1)
00076 //---------------------------------------------------------------------------
00077
00078 #endif // KERNEL_TIMERS_TICKLESS
00079
00080 //---------------------------------------------------------------------------
00091 typedef void (*TimerCallback_t)(Thread *pclOwner_, void *pvData_);
00092
00093 //---------------------------------------------------------------------------
00094 class TimerList;
00095 class TimerScheduler;
00096 class Quantum;
00102 class Timer : public LinkListNode
00103 {
00104 public:
00108     Timer() { }
00109
00113     void Init() { ClearNode(); m_u32Interval = 0;
      m_u32TimerTolerance = 0; m_u32TimeLeft = 0;
      m_u8Flags = 0; }
00114
00124     void Start( bool bRepeat_, uint32_t u32IntervalMs_, TimerCallback_t pfCallback_,
      void *pvData_ );
00125
00137     void Start( bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_,
      TimerCallback_t pfCallback_, void *pvData_ );
00138
00143     void Stop();
00144
00154     void SetFlags (uint8_t u8Flags_) { m_u8Flags = u8Flags_; }
00155
00163     void SetCallback( TimerCallback_t pfCallback_){
      m_pfCallback = pfCallback_; }
00164
00172     void SetData( void *pvData_ ){ m_pvData = pvData_; }
00173
00182     void SetOwner( Thread *pclOwner_){ m_pclOwner = pclOwner_; }
00183
00191     void SetIntervalTicks(uint32_t u32Ticks_);
00192
00200     void SetIntervalSeconds(uint32_t u32Seconds_);
00201
00202
00203     uint32_t GetInterval()  { return m_u32Interval; }
00204
00212     void SetIntervalMSeconds(uint32_t u32MSeconds_);
00213
00221     void SetIntervalUSeconds(uint32_t u32USeconds_);
00222
00232     void SetTolerance(uint32_t u32Ticks_);
00233
00234 private:
```

```
00235
00236     friend class TimerList;
00237
00239     uint8_t m_u8Flags;
00240
00242     TimerCallback_t m_pfCallback;
00243
00245     uint32_t m_u32Interval;
00246
00248     uint32_t m_u32TimeLeft;
00249
00251     uint32_t m_u32TimerTolerance;
00252
00254     Thread  *m_pclOwner;
00255
00257     void    *m_pvData;
00258 };
00259
00260 #endif // KERNEL_USE_TIMERS
00261
00262 #endif
```

15.93 /home/vm/mark3/trunk/embedded/kernel/public/timerlist.h File Reference

Timer list declarations.

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

Classes

- class TimerList

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

15.93.1 Detailed Description

Timer list declarations.

These classes implements a linked list of timer objects attached to the global kernel timer scheduler.

Definition in file timerlist.h.

15.94 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 =========================================================================== */
00024 #ifndef __TIMERLIST_H__
00025 #define __TIMERLIST_H__
00026
00027 #include "kerneltypes.h"
00028 #include "mark3cfg.h"
00029
00030 #include "timer.h"
00031 #if KERNEL_USE_TIMERS
00032
00033 //---------------------------------------------------------------------------
```

```
00037 class TimerList : public DoubleLinkList
00038 {
00039 public:
00046     void Init();
00047 
00055     void Add(Timer *pclListNode_);
00056 
00064     void Remove(Timer *pclListNode_);
00065 
00072     void Process();
00073 
00074 private:
00076     uint32_t m_u32NextWakeup;
00077 
00079     bool m_bTimerActive;
00080 };
00081 
00082 #endif // KERNEL_USE_TIMERS
00083 
00084 #endif
```

15.95 /home/vm/mark3/trunk/embedded/kernel/public/timerscheduler.h File Reference

Timer scheduler declarations.

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

Classes

- class TimerScheduler

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

15.95.1 Detailed Description

Timer scheduler declarations.

Definition in file timerscheduler.h.

15.96 timerscheduler.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 __TIMERSCHEDULER_H__
00022 #define __TIMERSCHEDULER_H__
00023 
00024 #include "kerneltypes.h"
00025 #include "mark3cfg.h"
00026 
00027 #include "ll.h"
00028 #include "timer.h"
00029 #include "timerlist.h"
00030 
```

```
00031 #if KERNEL_USE_TIMERS
00032
00033 //---------------------------------------------------------------------------
00038 class TimerScheduler
00039 {
00040 public:
00047     static void Init() { m_clTimerList.Init(); }
00048
00057     static void Add(Timer *pclListNode_)
00058         {m_clTimerList.Add(pclListNode_); }
00059
00068     static void Remove(Timer *pclListNode_)
00069         {m_clTimerList.Remove(pclListNode_); }
00070
00079     static void Process() {m_clTimerList.Process();}
00080 private:
00081
00083     static TimerList m_clTimerList;
00084 };
00085
00086 #endif //KERNEL_USE_TIMERS
00087
00088 #endif //__TIMERSCHEDULER_H__
00089
```

15.97 /home/vm/mark3/trunk/embedded/kernel/public/tracebuffer.h File Reference

Kernel trace buffer class declaration.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "writebuf16.h"
```

15.97.1 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.

15.98 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 uint16_t Increment();
00056 
00065     static void Write( uint16_t *pu16Data_, uint16_t u16Size_ );
00066 
00075     void SetCallback( WriteBufferCallback pfCallback_ )
00076         { m_clBuffer.SetCallback( pfCallback_ ); }
00077 private:
00078 
00079     static WriteBuffer16 m_clBuffer;
00080     static volatile uint16_t m_u16Index;
00081     static uint16_t m_au16Buffer[ (TRACE_BUFFER_SIZE / sizeof( uint16_t )) ];
00082 };
00083 
00084 #endif //KERNEL_USE_DEBUG
00085 
00086 #endif
```

15.99 /home/vm/mark3/trunk/embedded/kernel/public/writebuf16.h File Reference

Thread-safe circular buffer implementation with 16-bit elements.

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

15.99.1 Detailed Description

Thread-safe circular buffer implementation with 16-bit elements.

Definition in file writebuf16.h.

15.100 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)( uint16_t *pu16Data_, uint16_t u16Size_ );
00033 
00040 class WriteBuffer16
00041 {
00042 public:
00053     void SetBuffers( uint16_t *pu16Data_, uint16_t u16Size_ )
00054     {
00055         m_pu16Data = pu16Data_;
00056         m_u16Size = u16Size_;
00057         m_u16Head = 0;
00058         m_u16Tail = 0;
00059     }
00060 
00072     void SetCallback( WriteBufferCallback pfCallback_ )
00073         { m_pfCallback = pfCallback_; }
```

```
00074
00083     void WriteData( uint16_t *pu16Buf_, uint16_t u16Len_ );
00084
00094     void WriteVector( uint16_t **ppu16Buf_, uint16_t *pu16Len_, uint8_t u8Count_);
00095
00096 private:
00097     uint16_t *m_pu16Data;
00098
00099     volatile uint16_t m_u16Size;
00100     volatile uint16_t m_u16Head;
00101     volatile uint16_t m_u16Tail;
00102
00103     WriteBufferCallback m_pfCallback;
00104 };
00105 #endif
00106
00107 #endif
```

15.101 /home/vm/mark3/trunk/embedded/kernel/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 "kernelaware.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

Functions

- static void QuantumCallback (Thread *pclThread_, void *pvData_)

  QuantumCallback.

15.101.1 Detailed Description

Thread Quantum Implementation for Round-Robin Scheduling.

Definition in file quantum.cpp.

15.101.2 Function Documentation

15.101.2.1 static void QuantumCallback (Thread * *pclThread_*, void * *pvData_*) [static]

QuantumCallback.

This is the timer callback that is invoked whenever a thread has exhausted its current execution quantum and a new thread must be chosen from within the same priority level.

Parameters

*pclThread_*	Pointer to the thread currently executing
*pvData_*	Unused in this context.

Definition at line 62 of file quantum.cpp.

15.102 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 "kernelaware.h"
00029
00030 #define _CAN_HAS_DEBUG
00031 //--[Autogenerated - Do Not Modify]-------------------------------------------
00032 #include "dbg_file_list.h"
00033 #include "buffalogger.h"
00034 #if defined(DBG_FILE)
00035 # error "Debug logging file token already defined!  Bailing."
00036 #else
00037 # define DBG_FILE _DBG___KERNEL_QUANTUM_CPP
00038 #endif
00039 //--[End Autogenerated content]-----------------------------------------------
00040 #include "kerneldebug.h"
00041
00042 #if KERNEL_USE_QUANTUM
00043
00044 //---------------------------------------------------------------------------
00045 static volatile bool bAddQuantumTimer;  // Indicates that a timer add is pending
00046
00047 //---------------------------------------------------------------------------
00048 Timer   Quantum::m_clQuantumTimer; // The global timernodelist_t object
00049 bool Quantum::m_bActive;
00050 bool Quantum::m_bInTimer;
00051 //---------------------------------------------------------------------------
00062 static void QuantumCallback(Thread *pclThread_, void *pvData_)
00063 {
00064     // Validate thread pointer, check that source/destination match (it's
00065     // in its real priority list).  Also check that this thread was part of
00066     // the highest-running priority level.
00067     if (pclThread_->GetPriority() >= Scheduler::GetCurrentThread()->
      GetPriority())
00068     {
00069         if (pclThread_->GetCurrent()->GetHead() != pclThread_->
      GetCurrent()->GetTail() )
00070         {
00071             bAddQuantumTimer = true;
00072             pclThread_->GetCurrent()->PivotForward();
00073         }
00074     }
00075 }
00076
00077 //---------------------------------------------------------------------------
00078 void Quantum::SetTimer(Thread *pclThread_)
00079 {
00080     m_clQuantumTimer.SetIntervalMSeconds(pclThread_->
      GetQuantum());
00081     m_clQuantumTimer.SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00082     m_clQuantumTimer.SetData(NULL);
00083     m_clQuantumTimer.SetCallback((TimerCallback_t)
      QuantumCallback);
00084     m_clQuantumTimer.SetOwner(pclThread_);
00085 }
00086
00087 //---------------------------------------------------------------------------
00088 void Quantum::AddThread(Thread *pclThread_)
00089 {
00090     if (m_bActive
00091 #if KERNEL_USE_IDLE_FUNC
00092             || (pclThread_ == Kernel::GetIdleThread())
00093 #endif
00094        )
00095     {
00096         return;
00097     }
00098
```

```
00099     // If this is called from the timer callback, queue a timer add...
00100     if (m_bInTimer)
00101     {
00102         bAddQuantumTimer = true;
00103         return;
00104     }
00105 
00106     // If this isn't the only thread in the list.
00107     if ( pclThread_->GetCurrent()->GetHead() !=
00108            pclThread_->GetCurrent()->GetTail() )
00109     {
00110         Quantum::SetTimer(pclThread_);
00111         TimerScheduler::Add(&m_clQuantumTimer);
00112         m_bActive = 1;
00113     }
00114 }
00115 
00116 //---------------------------------------------------------------------------
00117 void Quantum::RemoveThread(void)
00118 {
00119     if (!m_bActive)
00120     {
00121         return;
00122     }
00123 
00124     // Cancel the current timer
00125     TimerScheduler::Remove(&m_clQuantumTimer);
00126     m_bActive = 0;
00127 }
00128 
00129 //---------------------------------------------------------------------------
00130 void Quantum::UpdateTimer(void)
00131 {
00132     // If we have to re-add the quantum timer (more than 2 threads at the
00133     // high-priority level...)
00134     if (bAddQuantumTimer)
00135     {
00136         // Trigger a thread yield - this will also re-schedule the
00137         // thread *and* reset the round-robin scheduler.
00138         Thread::Yield();
00139         bAddQuantumTimer = false;
00140     }
00141 }
00142 
00143 #endif //KERNEL_USE_QUANTUM
```

15.103 /home/vm/mark3/trunk/embedded/kernel/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.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

Variables

- volatile Thread * g_pclNext

  Pointer to the currently-chosen next-running thread.

- Thread * g_pclCurrent

  Pointer to the currently-running thread.

- static const uint8_t aucCLZ [16] ={255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3}

  This implements a 4-bit "Count-leading-zeros" operation using a RAM-based lookup table.

15.103.1 Detailed Description

Strict-Priority + Round-Robin thread scheduler implementation.

Definition in file scheduler.cpp.

15.103.2 Variable Documentation

15.103.2.1 const uint8_t aucCLZ[16] ={255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3} [static]

This implements a 4-bit "Count-leading-zeros" operation using a RAM-based lookup table.

It is used to efficiently perform a CLZ operation under the assumption that a native CLZ instruction is unavailable. This table is further optimized to provide a 0xFF result in the event that the index value is itself zero, allowing u16 to quickly identify whether or not subsequent 4-bit LUT operations are required to complete the scheduling process.

Definition at line 61 of file scheduler.cpp.

15.104 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.h"
00028
00029 #define _CAN_HAS_DEBUG
00030 //--[Autogenerated - Do Not Modify]-------------------------------------
00031 #include "dbg_file_list.h"
00032 #include "buffalogger.h"
00033 #if defined(DBG_FILE)
00034 # error "Debug logging file token already defined!  Bailing."
00035 #else
00036 # define DBG_FILE _DBG___KERNEL_SCHEDULER_CPP
00037 #endif
00038 //--[End Autogenerated content]-----------------------------------------
00039
00040 #include "kerneldebug.h"
00041 volatile Thread *g_pclNext;
00042 Thread *g_pclCurrent;
00043
00044 //---------------------------------------------------------------------------
00045 bool Scheduler::m_bEnabled;
00046 bool Scheduler::m_bQueuedSchedule;
00047
00048 ThreadList Scheduler::m_clStopList;
00049 ThreadList Scheduler::m_aclPriorities[
      NUM_PRIORITIES];
00050 uint8_t Scheduler::m_u8PriFlag;
00051
00052 //---------------------------------------------------------------------------
00061 static const uint8_t aucCLZ[16] ={255,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3};
00062
00063 //---------------------------------------------------------------------------
00064 void Scheduler::Init()
00065 {
00066     m_u8PriFlag = 0;
00067     for (int i = 0; i < NUM_PRIORITIES; i++)
00068     {
00069         m_aclPriorities[i].SetPriority(i);
00070         m_aclPriorities[i].SetFlagPointer(&
      m_u8PriFlag);
```

```
00071     }
00072     m_bQueuedSchedule = false;
00073 }
00074 
00075 //---------------------------------------------------------------------------
00076 void Scheduler::Schedule()
00077 {
00078     uint8_t u8Pri = 0;
00079 
00080     // Figure out what priority level has ready tasks (8 priorities max)
00081     // To do this, we apply our current active-thread bitmap (m_u8PriFlag)
00082     // and perform a CLZ on the upper four bits.  If no tasks are found
00083     // in the higher priority bits, search the lower priority bits.  This
00084     // also assumes that we always have the idle thread ready-to-run in
00085     // priority level zero.
00086     u8Pri = aucCLZ[m_u8PriFlag >> 4 ];
00087     if (u8Pri == 0xFF)
00088     {
00089         u8Pri = aucCLZ[m_u8PriFlag & 0x0F];
00090     }
00091     else
00092     {
00093         u8Pri += 4;
00094     }
00095 
00096 #if KERNEL_USE_IDLE_FUNC
00097     if (u8Pri == 0xFF)
00098     {
00099         // There aren't any active threads at all - set g_pclNext to IDLE
00100         g_pclNext = Kernel::GetIdleThread();
00101     }
00102     else
00103 #endif
00104     {
00105         // Get the thread node at this priority.
00106         g_pclNext = (Thread*)( m_aclPriorities[u8Pri].GetHead() );
00107     }
00108     KERNEL_TRACE_1( "Next Thread: %d\n", (uint16_t)((Thread*)g_pclNext)->GetID() );
00109 
00110 }
00111 
00112 //---------------------------------------------------------------------------
00113 void Scheduler::Add(Thread *pclThread_)
00114 {
00115     m_aclPriorities[pclThread_->GetPriority()].Add(pclThread_);
00116 }
00117 
00118 //---------------------------------------------------------------------------
00119 void Scheduler::Remove(Thread *pclThread_)
00120 {
00121     m_aclPriorities[pclThread_->GetPriority()].Remove(pclThread_);
00122 }
00123 
00124 //---------------------------------------------------------------------------
00125 bool Scheduler::SetScheduler(bool bEnable_)
00126 {
00127     bool bRet ;
00128     CS_ENTER();
00129     bRet = m_bEnabled;
00130     m_bEnabled = bEnable_;
00131     // If there was a queued scheduler evevent, dequeue and trigger an
00132     // immediate Yield
00133     if (m_bEnabled && m_bQueuedSchedule)
00134     {
00135         m_bQueuedSchedule = false;
00136         Thread::Yield();
00137     }
00138     CS_EXIT();
00139     return bRet;
00140 }
```

15.105 /home/vm/mark3/trunk/embedded/kernel/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 "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

Functions

- static void ThreadSleepCallback (Thread *pclOwner_, void *pvData_)

  This callback is used to wake up a thread once the interval has expired.

15.105.1 Detailed Description

Platform-Independent thread class Definition.

Definition in file thread.cpp.

15.106 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
00033 #define _CAN_HAS_DEBUG
00034 //--[Autogenerated - Do Not Modify]-------------------------------------
00035 #include "dbg_file_list.h"
00036 #include "buffalogger.h"
00037 #if defined(DBG_FILE)
00038 # error "Debug logging file token already defined!  Bailing."
00039 #else
00040 # define DBG_FILE _DBG___KERNEL_THREAD_CPP
00041 #endif
00042 //--[End Autogenerated content]-----------------------------------------
00043
00044 #include "kerneldebug.h"
00045 //---------------------------------------------------------------------------
00046 void Thread::Init(  K_WORD *pwStack_,
00047                 uint16_t u16StackSize_,
00048                 uint8_t u8Priority_,
00049                 ThreadEntry_t pfEntryPoint_,
00050                 void *pvArg_ )
```

```
00051 {
00052     static uint8_t u8ThreadID = 0;
00053 
00054     KERNEL_ASSERT( pwStack_ );
00055     KERNEL_ASSERT( pfEntryPoint_ );
00056 
00057     ClearNode();
00058 
00059     m_u8ThreadID = u8ThreadID++;
00060 
00061     KERNEL_TRACE_1( "Stack Size: %d", u16StackSize_ );
00062     KERNEL_TRACE_1( "Thread Pri: %d", (uint8_t)u8Priority_ );
00063     KERNEL_TRACE_1( "Thread Id: %d", (uint16_t)m_u8ThreadID );
00064     KERNEL_TRACE_1( "Entrypoint: %x", (uint16_t)pfEntryPoint_ );
00065 
00066     // Initialize the thread parameters to their initial values.
00067     m_pwStack = pwStack_;
00068     m_pwStackTop = TOP_OF_STACK(pwStack_, u16StackSize_);
00069 
00070     m_u16StackSize = u16StackSize_;
00071 
00072 #if KERNEL_USE_QUANTUM
00073     m_u16Quantum = THREAD_QUANTUM_DEFAULT;
00074 #endif
00075 
00076     m_u8Priority = u8Priority_ ;
00077     m_u8CurPriority = m_u8Priority;
00078     m_pfEntryPoint = pfEntryPoint_;
00079     m_pvArg = pvArg_;
00080     m_eState = THREAD_STATE_STOP;
00081 
00082 #if KERNEL_USE_THREADNAME
00083     m_szName = NULL;
00084 #endif
00085 #if KERNEL_USE_TIMERS
00086     m_clTimer.Init();
00087 #endif
00088 
00089     // Call CPU-specific stack initialization
00090     ThreadPort::InitStack(this);
00091 
00092     // Add to the global "stop" list.
00093     CS_ENTER();
00094     m_pclOwner = Scheduler::GetThreadList(
      m_u8Priority);
00095     m_pclCurrent = Scheduler::GetStopList();
00096     m_pclCurrent->Add(this);
00097     CS_EXIT();
00098 }
00099 
00100 //---------------------------------------------------------------------------
00101 void Thread::Start(void)
00102 {
00103     // Remove the thread from the scheduler's "stopped" list, and add it
00104     // to the scheduler's ready list at the proper priority.
00105     KERNEL_TRACE_1( "Starting Thread %d", (uint16_t)m_u8ThreadID );
00106 
00107     CS_ENTER();
00108     Scheduler::GetStopList()->Remove(this);
00109     Scheduler::Add(this);
00110     m_pclOwner = Scheduler::GetThreadList(
      m_u8Priority);
00111     m_pclCurrent = m_pclOwner;
00112     m_eState = THREAD_STATE_READY;
00113 
00114 #if KERNEL_USE_QUANTUM
00115     if (GetCurPriority() >= Scheduler::GetCurrentThread()->
      GetCurPriority())
00116     {
00117         // Deal with the thread Quantum
00118         Quantum::RemoveThread();
00119         Quantum::AddThread(this);
00120     }
00121 #endif
00122 
00123     if (Kernel::IsStarted())
00124     {
00125         if (GetCurPriority() >= Scheduler::GetCurrentThread()->
      GetCurPriority())
00126         {
00127             Thread::Yield();
00128         }
00129     }
00130     CS_EXIT();
00131 }
00132 
00133 //---------------------------------------------------------------------------
```

```
00134 void Thread::Stop()
00135 {
00136     bool bReschedule = 0;
00137 
00138     CS_ENTER();
00139 
00140     // If a thread is attempting to stop itself, ensure we call the scheduler
00141     if (this == Scheduler::GetCurrentThread())
00142     {
00143         bReschedule = true;
00144     }
00145 
00146     // Add this thread to the stop-list (removing it from active scheduling)
00147     // Remove the thread from scheduling
00148     if (m_eState == THREAD_STATE_READY)
00149     {
00150         Scheduler::Remove(this);
00151     }
00152     else if (m_eState == THREAD_STATE_BLOCKED)
00153     {
00154         m_pclCurrent->Remove(this);
00155     }
00156 
00157     m_pclOwner = Scheduler::GetStopList();
00158     m_pclCurrent = m_pclOwner;
00159     m_pclOwner->Add(this);
00160     m_eState = THREAD_STATE_STOP;
00161 
00162 #if KERNEL_USE_TIMERS
00163     // Just to be safe - attempt to remove the thread's timer
00164     // from the timer-scheduler (does no harm if it isn't
00165     // in the timer-list)
00166     TimerScheduler::Remove(&m_clTimer);
00167 #endif
00168 
00169     CS_EXIT();
00170 
00171     if (bReschedule)
00172     {
00173         Thread::Yield();
00174     }
00175 }
00176 
00177 #if KERNEL_USE_DYNAMIC_THREADS
00178 //---------------------------------------------------------------------------
00179 void Thread::Exit()
00180 {
00181     bool bReschedule = 0;
00182 
00183     KERNEL_TRACE_1( "Exit Thread %d", m_u8ThreadID );
00184 
00185     CS_ENTER();
00186 
00187     // If this thread is the actively-running thread, make sure we run the
00188     // scheduler again.
00189     if (this == Scheduler::GetCurrentThread())
00190     {
00191         bReschedule = 1;
00192     }
00193 
00194     // Remove the thread from scheduling
00195     if (m_eState == THREAD_STATE_READY)
00196     {
00197         Scheduler::Remove(this);
00198     }
00199     else if (m_eState == THREAD_STATE_BLOCKED)
00200     {
00201         m_pclCurrent->Remove(this);
00202     }
00203 
00204     m_pclCurrent = 0;
00205     m_pclOwner = 0;
00206     m_eState = THREAD_STATE_EXIT;
00207 
00208     // We've removed the thread from scheduling, but interrupts might
00209     // trigger checks against this thread's currently priority before
00210     // we get around to scheduling new threads.  As a result, set the
00211     // priority to idle to ensure that we always wind up scheduling
00212     // new threads.
00213     m_u8CurPriority = 0;
00214     m_u8Priority = 0;
00215 
00216 #if KERNEL_USE_TIMERS
00217     // Just to be safe - attempt to remove the thread's timer
00218     // from the timer-scheduler (does no harm if it isn't
00219     // in the timer-list)
00220     TimerScheduler::Remove(&m_clTimer);
```

```
00221 #endif
00222
00223     CS_EXIT();
00224
00225     if (bReschedule)
00226     {
00227         // Choose a new "next" thread if we must
00228         Thread::Yield();
00229     }
00230 }
00231 #endif
00232
00233 #if KERNEL_USE_SLEEP
00234 //---------------------------------------------------------------------------
00236 static void ThreadSleepCallback( Thread *pclOwner_, void *pvData_ )
00237 {
00238     Semaphore *pclSemaphore = static_cast<Semaphore*>(pvData_);
00239     // Post the semaphore, which will wake the sleeping thread.
00240     pclSemaphore->Post();
00241 }
00242
00243 //---------------------------------------------------------------------------
00244 void Thread::Sleep(uint32_t u32TimeMs_)
00245 {
00246     Semaphore clSemaphore;
00247     Timer *pclTimer = g_pclCurrent->GetTimer();
00248
00249     // Create a semaphore that this thread will block on
00250     clSemaphore.Init(0, 1);
00251
00252     // Create a one-shot timer that will call a callback that posts the
00253     // semaphore, waking our thread.
00254     pclTimer->Init();
00255     pclTimer->SetIntervalMSeconds(u32TimeMs_);
00256     pclTimer->SetCallback(ThreadSleepCallback);
00257     pclTimer->SetData((void*)&clSemaphore);
00258     pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00259
00260     // Add the new timer to the timer scheduler, and block the thread
00261     TimerScheduler::Add(pclTimer);
00262     clSemaphore.Pend();
00263 }
00264
00265 //---------------------------------------------------------------------------
00266 void Thread::USleep(uint32_t u32TimeUs_)
00267 {
00268     Semaphore clSemaphore;
00269     Timer *pclTimer = g_pclCurrent->GetTimer();
00270
00271     // Create a semaphore that this thread will block on
00272     clSemaphore.Init(0, 1);
00273
00274     // Create a one-shot timer that will call a callback that posts the
00275     // semaphore, waking our thread.
00276     pclTimer->Init();
00277     pclTimer->SetIntervalUSeconds(u32TimeUs_);
00278     pclTimer->SetCallback(ThreadSleepCallback);
00279     pclTimer->SetData((void*)&clSemaphore);
00280     pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00281
00282     // Add the new timer to the timer scheduler, and block the thread
00283     TimerScheduler::Add(pclTimer);
00284     clSemaphore.Pend();
00285 }
00286 #endif // KERNEL_USE_SLEEP
00287
00288 //---------------------------------------------------------------------------
00289 uint16_t Thread::GetStackSlack()
00290 {
00291     uint16_t u16Count = 0;
00292
00293     CS_ENTER();
00294
00296     for (u16Count = 0; u16Count < m_u16StackSize; u16Count++)
00297     {
00298         if (m_pwStack[u16Count] != 0xFF)
00299         {
00300             break;
00301         }
00302     }
00303
00304     CS_EXIT();
00305
00306     return u16Count;
00307 }
00308
00309 //---------------------------------------------------------------------------
```

```
00310 void Thread::Yield()
00311 {
00312     CS_ENTER();
00313     // Run the scheduler
00314     if (Scheduler::IsEnabled())
00315     {
00316         Scheduler::Schedule();
00317 
00318         // Only switch contexts if the new task is different than the old task
00319         if (Scheduler::GetCurrentThread() !=
      Scheduler::GetNextThread())
00320         {
00321 #if KERNEL_USE_QUANTUM
00322             // new thread scheduled.  Stop current quantum timer (if it exists),
00323             // and restart it for the new thread (if required).
00324             Quantum::RemoveThread();
00325             Quantum::AddThread((Thread*)g_pclNext);
00326 #endif
00327             Thread::ContextSwitchSWI();
00328         }
00329     }
00330     else
00331     {
00332         Scheduler::QueueScheduler();
00333     }
00334 
00335     CS_EXIT();
00336 }
00337 
00338 //---------------------------------------------------------------------------
00339 void Thread::SetPriorityBase(uint8_t u8Priority_)
00340 {
00341      GetCurrent()->Remove(this);
00342 
00343      SetCurrent(Scheduler::GetThreadList(
      m_u8Priority));
00344 
00345      GetCurrent()->Add(this);
00346 }
00347 
00348 //---------------------------------------------------------------------------
00349 void Thread::SetPriority(uint8_t u8Priority_)
00350 {
00351     bool bSchedule = 0;
00352 
00353     CS_ENTER();
00354     // If this is the currently running thread, it's a good idea to reschedule
00355     // Or, if the new priority is a higher priority than the current thread's.
00356     if ((g_pclCurrent == this) || (u8Priority_ > g_pclCurrent->
      GetPriority()))
00357     {
00358         bSchedule = 1;
00359     }
00360     Scheduler::Remove(this);
00361     CS_EXIT();
00362 
00363     m_u8CurPriority = u8Priority_;
00364     m_u8Priority = u8Priority_;
00365 
00366     CS_ENTER();
00367     Scheduler::Add(this);
00368     CS_EXIT();
00369 
00370     if (bSchedule)
00371     {
00372         if (Scheduler::IsEnabled())
00373         {
00374             CS_ENTER();
00375             Scheduler::Schedule();
00376     #if KERNEL_USE_QUANTUM
00377             // new thread scheduled.  Stop current quantum timer (if it exists),
00378             // and restart it for the new thread (if required).
00379             Quantum::RemoveThread();
00380             Quantum::AddThread((Thread*)g_pclNext);
00381     #endif
00382             CS_EXIT();
00383             Thread::ContextSwitchSWI();
00384         }
00385         else
00386         {
00387             Scheduler::QueueScheduler();
00388         }
00389     }
00390 }
00391 
00392 //---------------------------------------------------------------------------
00393 void Thread::InheritPriority(uint8_t u8Priority_)
```

```
00394 {
00395     SetOwner(Scheduler::GetThreadList(u8Priority_));
00396     m_u8CurPriority = u8Priority_;
00397 }
00398
00399 //---------------------------------------------------------------------------
00400 void Thread::ContextSwitchSWI()
00401 {
00402     // Call the context switch interrupt if the scheduler is enabled.
00403     if (Scheduler::IsEnabled() == 1)
00404     {
00405         KERNEL_TRACE_1( "Context switch to Thread %d", (uint16_t)((Thread*)
      g_pclNext)->GetID() );
00406         KernelSWI::Trigger();
00407     }
00408 }
00409
00410 #if KERNEL_USE_TIMEOUTS
00411 //---------------------------------------------------------------------------
00412 Timer *Thread::GetTimer()                       { return &
      m_clTimer; }
00413
00414 //---------------------------------------------------------------------------
00415 void Thread::SetExpired( bool bExpired_ )     { m_bExpired = bExpired_; }
00416
00417 //---------------------------------------------------------------------------
00418 bool Thread::GetExpired()                       { return
      m_bExpired; }
00419 #endif
00420
00421 #if KERNEL_USE_IDLE_FUNC
00422 //---------------------------------------------------------------------------
00423 void Thread::InitIdle( void )
00424 {
00425     ClearNode();
00426
00427     m_u8Priority = 0;
00428     m_u8CurPriority = 0;
00429     m_pfEntryPoint = 0;
00430     m_pvArg = 0;
00431     m_u8ThreadID = 255;
00432     m_eState = THREAD_STATE_READY;
00433 #if KERNEL_USE_THREADNAME
00434     m_szName = "IDLE";
00435 #endif
00436 }
00437 #endif
```

15.107 /home/vm/mark3/trunk/embedded/kernel/threadlist.cpp File Reference

Thread linked-list definitions.

```
#include "kerneltypes.h"
#include "ll.h"
#include "threadlist.h"
#include "thread.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

15.107.1 Detailed Description

Thread linked-list definitions.

Definition in file threadlist.cpp.

15.108 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
00027 #define _CAN_HAS_DEBUG
00028 //--[Autogenerated - Do Not Modify]-------------------------------------------
00029 #include "dbg_file_list.h"
00030 #include "buffalogger.h"
00031 #if defined(DBG_FILE)
00032 # error "Debug logging file token already defined!  Bailing."
00033 #else
00034 # define DBG_FILE _DBG___KERNEL_THREADLIST_CPP
00035 #endif
00036 //--[End Autogenerated content]-----------------------------------------------
00037 #include "kerneldebug.h"
00038
00039 //---------------------------------------------------------------------------
00040 void ThreadList::SetPriority(uint8_t u8Priority_)
00041 {
00042     m_u8Priority = u8Priority_;
00043 }
00044
00045 //---------------------------------------------------------------------------
00046 void ThreadList::SetFlagPointer( uint8_t *pu8Flag_)
00047 {
00048     m_pu8Flag = pu8Flag_;
00049 }
00050
00051 //---------------------------------------------------------------------------
00052 void ThreadList::Add(LinkListNode *node_) {
00053     CircularLinkList::Add(node_);
00054     CircularLinkList::PivotForward();
00055
00056     // We've specified a bitmap for this threadlist
00057     if (m_pu8Flag)
00058     {
00059         // Set the flag for this priority level
00060         *m_pu8Flag |= (1 << m_u8Priority);
00061     }
00062 }
00063
00064 //---------------------------------------------------------------------------
00065 void ThreadList::Add(LinkListNode *node_, uint8_t *pu8Flag_, uint8_t u8Priority_
      ) {
00066     // Set the threadlist's priority level, flag pointer, and then add the
00067     // thread to the threadlist
00068     SetPriority(u8Priority_);
00069     SetFlagPointer(pu8Flag_);
00070     Add(node_);
00071 }
00072
00073 //---------------------------------------------------------------------------
00074 void ThreadList::Remove(LinkListNode *node_) {
00075     // Remove the thread from the list
00076     CircularLinkList::Remove(node_);
00077
00078     // If the list is empty...
00079     if (!m_pstHead)
00080     {
00081         // Clear the bit in the bitmap at this priority level
00082         if (m_pu8Flag)
00083         {
00084             *m_pu8Flag &= ~(1 << m_u8Priority);
00085         }
00086     }
00087 }
00088
00089 //---------------------------------------------------------------------------
00090 Thread *ThreadList::HighestWaiter()
00091 {
00092     Thread *pclTemp = static_cast<Thread*>(GetHead());
00093     Thread *pclChosen = pclTemp;
00094
00095     uint8_t u8MaxPri = 0;
00096
```

```
00097     // Go through the list, return the highest-priority thread in this list.
00098     while(1)
00099     {
00100         // Compare against current max-priority thread
00101         if (pclTemp->GetPriority() >= u8MaxPri)
00102         {
00103             u8MaxPri = pclTemp->GetPriority();
00104             pclChosen = pclTemp;
00105         }
00106 
00107         // Break out if this is the last thread in the list
00108         if (pclTemp == static_cast<Thread*>(GetTail()))
00109         {
00110             break;
00111         }
00112 
00113         pclTemp = static_cast<Thread*>(pclTemp->GetNext());
00114     }
00115     return pclChosen;
00116 }
```

15.109 /home/vm/mark3/trunk/embedded/kernel/timer.cpp File Reference

Timer implementations.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "timer.h"
#include "timerlist.h"
#include "timerscheduler.h"
#include "kerneltimer.h"
#include "threadport.h"
#include "quantum.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

15.109.1 Detailed Description

Timer implementations.

Definition in file timer.cpp.

15.110 timer.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 "timer.h"
00026 #include "timerlist.h"
00027 #include "timerscheduler.h"
00028 #include "kerneltimer.h"
00029 #include "threadport.h"
00030 #include "quantum.h"
00031 
```

```
00032 #define _CAN_HAS_DEBUG
00033 //--[Autogenerated - Do Not Modify]----------------------------------------
00034 #include "dbg_file_list.h"
00035 #include "buffalogger.h"
00036 #if defined(DBG_FILE)
00037 # error "Debug logging file token already defined!  Bailing."
00038 #else
00039 # define DBG_FILE _DBG___KERNEL_TIMER_CPP
00040 #endif
00041 //--[End Autogenerated content]----------------------------------------------
00042
00043 #include "kerneldebug.h"
00044
00045 #if KERNEL_USE_TIMERS
00046
00047 //---------------------------------------------------------------------------
00048 void Timer::Start( bool bRepeat_, uint32_t u32IntervalMs_,
      TimerCallback_t pfCallback_, void *pvData_ )
00049 {
00050     SetIntervalMSeconds(u32IntervalMs_);
00051     m_u32TimerTolerance = 0;
00052     m_pfCallback = pfCallback_;
00053     m_pvData = pvData_;
00054     if (!bRepeat_)
00055     {
00056         m_u8Flags = TIMERLIST_FLAG_ONE_SHOT;
00057     }
00058     else
00059     {
00060         m_u8Flags = 0;
00061     }
00062     m_pclOwner = Scheduler::GetCurrentThread();
00063     TimerScheduler::Add(this);
00064 }
00065
00066 //---------------------------------------------------------------------------
00067 void Timer::Start( bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_,
      TimerCallback_t pfCallback_, void *pvData_ )
00068 {
00069     m_u32TimerTolerance = MSECONDS_TO_TICKS(u32ToleranceMs_);
00070     Start(bRepeat_, u32IntervalMs_, pfCallback_, pvData_);
00071 }
00072
00073 //---------------------------------------------------------------------------
00074 void Timer::Stop()
00075 {
00076     TimerScheduler::Remove(this);
00077 }
00078
00079 //---------------------------------------------------------------------------
00080 void Timer::SetIntervalTicks( uint32_t u32Ticks_ )
00081 {
00082     m_u32Interval = u32Ticks_;
00083 }
00084
00085 //---------------------------------------------------------------------------
00087 //---------------------------------------------------------------------------
00088 void Timer::SetIntervalSeconds( uint32_t u32Seconds_)
00089 {
00090     m_u32Interval = SECONDS_TO_TICKS(u32Seconds_);
00091 }
00092
00093 //---------------------------------------------------------------------------
00094 void Timer::SetIntervalMSeconds( uint32_t u32MSeconds_)
00095 {
00096     m_u32Interval = MSECONDS_TO_TICKS(u32MSeconds_);
00097 }
00098
00099 //---------------------------------------------------------------------------
00100 void Timer::SetIntervalUSeconds( uint32_t u32USeconds_)
00101 {
00102     m_u32Interval = useCONDS_TO_TICKS(u32USeconds_);
00103 }
00104
00105 //---------------------------------------------------------------------------
00106 void Timer::SetTolerance(uint32_t u32Ticks_)
00107 {
00108     m_u32TimerTolerance = u32Ticks_;
00109 }
00110
00111 #endif
```

15.111 /home/vm/mark3/trunk/embedded/kernel/timerlist.cpp File Reference

Implements timer list processing algorithms, responsible for all timer tick and expiry logic.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "timerlist.h"
#include "kerneltimer.h"
#include "threadport.h"
#include "quantum.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

15.111.1 Detailed Description

Implements timer list processing algorithms, responsible for all timer tick and expiry logic.

Definition in file timerlist.cpp.

15.112 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 ===========================================================================*/
00023 #include "kerneltypes.h"
00024 #include "mark3cfg.h"
00025
00026 #include "timerlist.h"
00027 #include "kerneltimer.h"
00028 #include "threadport.h"
00029 #include "quantum.h"
00030
00031 #define _CAN_HAS_DEBUG
00032 //--[Autogenerated - Do Not Modify]-------------------------------------------
00033 #include "dbg_file_list.h"
00034 #include "buffalogger.h"
00035 #if defined(DBG_FILE)
00036 # error "Debug logging file token already defined!  Bailing."
00037 #else
00038 # define DBG_FILE _DBG___KERNEL_TIMERLIST_CPP
00039 #endif
00040 //--[End Autogenerated content]-----------------------------------------------
00041
00042 #include "kerneldebug.h"
00043
00044 #if KERNEL_USE_TIMERS
00045 //---------------------------------------------------------------------------
00046 TimerList TimerScheduler::m_clTimerList;
00047
00048 //---------------------------------------------------------------------------
00049 void TimerList::Init(void)
00050 {
00051     m_bTimerActive = 0;
00052     m_u32NextWakeup = 0;
00053 }
00054
00055 //---------------------------------------------------------------------------
00056 void TimerList::Add(Timer *pclListNode_)
00057 {
00058 #if KERNEL_TIMERS_TICKLESS
00059     bool bStart = 0;
```

```
00060     int32_t lDelta;
00061 #endif
00062
00063     CS_ENTER();
00064
00065 #if KERNEL_TIMERS_TICKLESS
00066     if (GetHead() == NULL)
00067     {
00068         bStart = 1;
00069     }
00070 #endif
00071
00072     pclListNode_->ClearNode();
00073     DoubleLinkList::Add(pclListNode_);
00074
00075     // Set the initial timer value
00076     pclListNode_->m_u32TimeLeft = pclListNode_->m_u32Interval;
00077
00078 #if KERNEL_TIMERS_TICKLESS
00079     if (!bStart)
00080     {
00081         // If the new interval is less than the amount of time remaining...
00082         lDelta = KernelTimer::TimeToExpiry() - pclListNode_->
    m_u32Interval;
00083
00084         if (lDelta > 0)
00085         {
00086             // Set the new expiry time on the timer.
00087             m_u32NextWakeup = KernelTimer::SubtractExpiry((
    uint32_t)lDelta);
00088         }
00089     }
00090     else
00091     {
00092         m_u32NextWakeup = pclListNode_->m_u32Interval;
00093         KernelTimer::SetExpiry(m_u32NextWakeup);
00094         KernelTimer::Start();
00095     }
00096 #endif
00097
00098     // Set the timer as active.
00099     pclListNode_->m_u8Flags |= TIMERLIST_FLAG_ACTIVE;
00100     CS_EXIT();
00101 }
00102
00103 //---------------------------------------------------------------------------
00104 void TimerList::Remove(Timer *pclLinkListNode_)
00105 {
00106     CS_ENTER();
00107
00108     DoubleLinkList::Remove(pclLinkListNode_);
00109
00110 #if KERNEL_TIMERS_TICKLESS
00111     if (this->GetHead() == NULL)
00112     {
00113         KernelTimer::Stop();
00114     }
00115 #endif
00116
00117     CS_EXIT();
00118 }
00119
00120 //---------------------------------------------------------------------------
00121 void TimerList::Process(void)
00122 {
00123 #if KERNEL_TIMERS_TICKLESS
00124     uint32_t u32NewExpiry;
00125     uint32_t u32Overtime;
00126     bool bContinue;
00127 #endif
00128
00129     Timer *pclNode;
00130     Timer *pclPrev;
00131
00132 #if KERNEL_USE_QUANTUM
00133     Quantum::SetInTimer();
00134 #endif
00135 #if KERNEL_TIMERS_TICKLESS
00136     // Clear the timer and its expiry time - keep it running though
00137     KernelTimer::ClearExpiry();
00138     do
00139     {
00140 #endif
00141         pclNode = static_cast<Timer*>(GetHead());
00142         pclPrev = NULL;
00143
00144 #if KERNEL_TIMERS_TICKLESS
```

```
00145         bContinue = 0;
00146         u32NewExpiry = MAX_TIMER_TICKS;
00147 #endif
00148 
00149         // Subtract the elapsed time interval from each active timer.
00150         while (pclNode)
00151         {
00152             // Active timers only...
00153             if (pclNode->m_u8Flags & TIMERLIST_FLAG_ACTIVE)
00154             {
00155                 // Did the timer expire?
00156 #if KERNEL_TIMERS_TICKLESS
00157                 if (pclNode->m_u32TimeLeft <= m_u32NextWakeup)
00158 #else
00159                 pclNode->m_u32TimeLeft--;
00160                 if (0 == pclNode->m_u32TimeLeft)
00161 #endif
00162                 {
00163                     // Yes - set the "callback" flag - we'll execute the callbacks later
00164                     pclNode->m_u8Flags |= TIMERLIST_FLAG_CALLBACK;
00165 
00166                     if (pclNode->m_u8Flags & TIMERLIST_FLAG_ONE_SHOT)
00167                     {
00168                         // If this was a one-shot timer, deactivate the timer.
00169                         pclNode->m_u8Flags |= TIMERLIST_FLAG_EXPIRED;
00170                         pclNode->m_u8Flags &= ~TIMERLIST_FLAG_ACTIVE;

00171                     }
00172                     else
00173                     {
00174                         // Reset the interval timer.
00176                         // I think we're good though...
00177                         pclNode->m_u32TimeLeft = pclNode->
      m_u32Interval;
00178 
00179 #if KERNEL_TIMERS_TICKLESS
00180                         // If the time remaining (plus the length of the tolerance interval)
00181                         // is less than the next expiry interval, set the next expiry interval.
00182                         uint32_t u32Tmp = pclNode->m_u32TimeLeft + pclNode->
      m_u32TimerTolerance;
00183 
00184                         if (u32Tmp < u32NewExpiry)
00185                         {
00186                             u32NewExpiry = u32Tmp;
00187                         }
00188 #endif
00189                     }
00190                 }
00191 #if KERNEL_TIMERS_TICKLESS
00192                 else
00193                 {
00194                     // Not expiring, but determine how int32_t to run the next timer interval for.
00195                     pclNode->m_u32TimeLeft -= m_u32NextWakeup;
00196                     if (pclNode->m_u32TimeLeft < u32NewExpiry)
00197                     {
00198                         u32NewExpiry = pclNode->m_u32TimeLeft;
00199                     }
00200                 }
00201 #endif
00202             }
00203             pclNode = static_cast<Timer*>(pclNode->GetNext());
00204         }
00205 
00206         // Process the expired timers callbacks.
00207         pclNode = static_cast<Timer*>(GetHead());
00208         while (pclNode)
00209         {
00210             pclPrev = NULL;
00211 
00212             // If the timer expired, run the callbacks now.
00213             if (pclNode->m_u8Flags & TIMERLIST_FLAG_CALLBACK)
00214             {
00215                 // Run the callback. these callbacks must be very fast...
00216                 pclNode->m_pfCallback( pclNode->m_pclOwner, pclNode->
      m_pvData );
00217                 pclNode->m_u8Flags &= ~TIMERLIST_FLAG_CALLBACK;
00218 
00219                 // If this was a one-shot timer, let's remove it.
00220                 if (pclNode->m_u8Flags & TIMERLIST_FLAG_ONE_SHOT)
00221                 {
00222                     pclPrev = pclNode;
00223                 }
00224             }
00225             pclNode = static_cast<Timer*>(pclNode->GetNext());
00226 
00227             // Remove one-shot-timers
00228             if (pclPrev)
```

```
00229             {
00230                 Remove(pclPrev);
00231             }
00232         }
00233 
00234 #if KERNEL_TIMERS_TICKLESS
00235         // Check to see how much time has elapsed since the time we
00236         // acknowledged the interrupt...
00237         u32Overtime = KernelTimer::GetOvertime();
00238 
00239         if( u32Overtime >= u32NewExpiry ) {
00240             m_u32NextWakeup = u32Overtime;
00241             bContinue = 1;
00242         }
00243 
00244     // If it's taken longer to go through this loop than would take u16 to
00245     // the next expiry, re-run the timing loop
00246 
00247     } while (bContinue);
00248 
00249     // This timer elapsed, but there's nothing more to do...
00250     // Turn the timer off.
00251     if (u32NewExpiry >= MAX_TIMER_TICKS)
00252     {
00253         KernelTimer::Stop();
00254     }
00255     else
00256     {
00257         // Update the timer with the new "Next Wakeup" value, plus whatever
00258         // overtime has accumulated since the last time we called this handler
00259 
00260         m_u32NextWakeup = KernelTimer::SetExpiry(u32NewExpiry +
      u32Overtime);
00261     }
00262 #endif
00263 #if KERNEL_USE_QUANTUM
00264     Quantum::ClearInTimer();
00265 #endif
00266 }
00267 
00268 
00269 #endif //KERNEL_USE_TIMERS
```

15.113 /home/vm/mark3/trunk/embedded/kernel/tracebuffer.cpp File Reference

Kernel trace buffer class definition.

```
#include "kerneltypes.h"
#include "tracebuffer.h"
#include "mark3cfg.h"
#include "writebuf16.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

15.113.1 Detailed Description

Kernel trace buffer class definition.

Definition in file tracebuffer.cpp.

15.114 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
00024 #define _CAN_HAS_DEBUG
00025 //--[Autogenerated - Do Not Modify]-----------------------------------------
00026 #include "dbg_file_list.h"
00027 #include "buffalogger.h"
00028 #if defined(DBG_FILE)
00029 # error "Debug logging file token already defined!  Bailing."
00030 #else
00031 # define DBG_FILE _DBG___KERNEL_TRACEBUFFER_CPP
00032 #endif
00033
00034 #include "kerneldebug.h"
00035
00036 //--[End Autogenerated content]---------------------------------------------
00037
00038 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00039 //---------------------------------------------------------------------------
00040 WriteBuffer16 TraceBuffer::m_clBuffer;
00041 volatile uint16_t TraceBuffer::m_u16Index;
00042 uint16_t TraceBuffer::m_au16Buffer[ (TRACE_BUFFER_SIZE/sizeof(uint16_t)) ];
00043
00044 //---------------------------------------------------------------------------
00045 void TraceBuffer::Init()
00046 {
00047     m_clBuffer.SetBuffers(m_au16Buffer, TRACE_BUFFER_SIZE/sizeof(uint16_t));
00048     m_u16Index = 0;
00049 }
00050
00051 //---------------------------------------------------------------------------
00052 uint16_t TraceBuffer::Increment()
00053 {
00054     return m_u16Index++;
00055 }
00056
00057 //---------------------------------------------------------------------------
00058 void TraceBuffer::Write( uint16_t *pu16Data_, uint16_t u16Size_ )
00059 {
00060     // Pipe the data directly to the circular buffer
00061     m_clBuffer.WriteData(pu16Data_, u16Size_);
00062 }
00063
00064 #endif
00065
```

15.115 /home/vm/mark3/trunk/embedded/kernel/writebuf16.cpp File Reference

16 bit circular buffer implementation with callbacks.

```
#include "kerneltypes.h"
#include "writebuf16.h"
#include "threadport.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
```

15.115.1 Detailed Description

16 bit circular buffer implementation with callbacks.

Definition in file writebuf16.cpp.

15.116 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 "threadport.h"
00023
00024 #define _CAN_HAS_DEBUG
00025 //--[Autogenerated - Do Not Modify]-------------------------------------------
00026 #include "dbg_file_list.h"
00027 #include "buffalogger.h"
00028 #if defined(DBG_FILE)
00029 # error "Debug logging file token already defined!  Bailing."
00030 #else
00031 # define DBG_FILE _DBG___KERNEL_WRITEBUF16_CPP
00032 #endif
00033 //--[End Autogenerated content]-----------------------------------------------
00034 #include "kerneldebug.h"
00035
00036 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00037
00038 //---------------------------------------------------------------------------
00039 void WriteBuffer16::WriteData( uint16_t *pu16Buf_, uint16_t u16Len_ )
00040 {
00041     uint16_t *apu16Buf[1];
00042     uint16_t au16Len[1];
00043
00044     apu16Buf[0] = pu16Buf_;
00045     au16Len[0] = u16Len_;
00046
00047     WriteVector( apu16Buf, au16Len, 1 );
00048 }
00049
00050 //---------------------------------------------------------------------------
00051 void WriteBuffer16::WriteVector( uint16_t **ppu16Buf_, uint16_t *pu16Len_, uint8_t u8Count_ )
00052 {
00053     uint16_t u16TempHead;
00054     uint8_t i;
00055     uint8_t j;
00056     uint16_t u16TotalLen = 0;
00057     bool bCallback = false;
00058     bool bRollover = false;
00059     // Update the head pointer synchronously, using a small
00060     // critical section in order to provide thread safety without
00061     // compromising on responsiveness by adding lots of extra
00062     // interrupt latency.
00063
00064     CS_ENTER();
00065
00066     u16TempHead = m_u16Head;
00067     {
00068         for (i = 0; i < u8Count_; i++)
00069         {
00070             u16TotalLen += pu16Len_[i];
00071         }
00072         m_u16Head = (u16TempHead + u16TotalLen) % m_u16Size;
00073     }
00074     CS_EXIT();
00075
00076     // Call the callback if we cross the 50% mark or rollover
00077     if (m_u16Head < u16TempHead)
00078     {
00079         if (m_pfCallback)
00080         {
00081             bCallback = true;
00082             bRollover = true;
00083         }
00084     }
00085     else if ((u16TempHead < (m_u16Size >> 1)) && (m_u16Head >= (m_u16Size >> 1)))
00086     {
00087         // Only trigger the callback if it's non-null
00088         if (m_pfCallback)
00089         {
00090             bCallback = true;
```

```
00091         }
00092     }
00093 
00094     // Are we going to roll-over?
00095     for (j = 0; j < u8Count_; j++)
00096     {
00097         uint16_t u16SegmentLength = pu16Len_[j];
00098         if (u16SegmentLength + u16TempHead >= m_u16Size)
00099         {
00100             // We need to two-part this... First part: before the rollover
00101             uint16_t u16TempLen;
00102             uint16_t *pu16Tmp = &m_pu16Data[ u16TempHead ];
00103             uint16_t *pu16Src = ppu16Buf_[j];
00104             u16TempLen = m_u16Size - u16TempHead;
00105             for (i = 0; i < u16TempLen; i++)
00106             {
00107                 *pu16Tmp++ = *pu16Src++;
00108             }
00109 
00110             // Second part: after the rollover
00111             u16TempLen = u16SegmentLength - u16TempLen;
00112             pu16[A-Z]mp = m_pu16Data;
00113             for (i = 0; i < u16TempLen; i++)
00114             {
00115                 *pu16Tmp++ = *pu16Src++;
00116             }
00117         }
00118         else
00119         {
00120             // No rollover - do the copy all at once.
00121             uint16_t *pu16Src = ppu16Buf_[j];
00122             uint16_t *pu16Tmp = &m_pu16Data[ u16TempHead ];
00123             for (uint16_t i = 0; i < u16SegmentLength; i++)
00124             {
00125                 *pu16Tmp++ = *pu16Src++;
00126             }
00127         }
00128     }
00129 
00130 
00131     // Call the callback if necessary
00132     if (bCallback)
00133     {
00134         if (bRollover)
00135         {
00136             // Rollover - process the back-half of the buffer
00137             m_pfCallback( &m_pu16Data[ m_u16Size >> 1], m_u16Size >> 1 );
00138         }
00139         else
00140         {
00141             // 50% point - process the front-half of the buffer
00142             m_pfCallback( m_pu16Data, m_u16Size >> 1);
00143         }
00144     }
00145 }
00146 
00147 #endif
```

Index