Mark3 Realtime Kernel

Generated by Doxygen 1.8.7

Wed Sep 21 2016 23:08:18

Contents

Chapter 1

The Mark3 Realtime Kernel
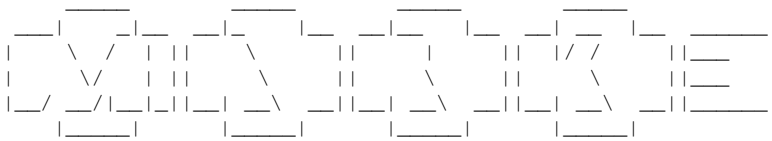
```
      _____        _____        _____        _____
  ___|    _|__   __|_         |__   __|__    |__   _____
 |      \  /   |  ||      \       ||       |      ||   |/ /     ||___    |
 |       \/    |  ||       \      ||         \      ||       \    ||___   |
 |__/  __/|__|_||__| __\  __||__| __\  __||__| __\  __||_____|
     |_____|       |_____|       |_____|       |_____|
--[Mark3 Realtime Platform]------------------------------------------------

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license for more information
```

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

It uses modern programming languages and concepts to minimize code duplication, and its object-oriented design enhances readibility. The API is simple – in six function calls, you can set up the kernel, initialize two 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:

- Native implementation in C++, with C-language bindings.

- 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, incluing Funkenstein's own flAVR.

Chapter 2

License

2.1 License

Chapter 3

Configuring The Mark3 Kernel

3.1   Overview

The Mark3 Kernel features a large number of compile-time options that can be set by the user. In this way, the user can build a custom OS kernel that provides only the necessary feature set required by the application, and reduce the code and data requirements of the kernel.

Care has been taken to ensure that all valid combinations of features can be enabled or disabled, barring direct dependencies.

When Mark3 is built, the various compile-time definitions are used to alter how the kernel is compiled, and include or exclude various bits and pieces in order to satisfy the requirements of the selected features. As a result, the kernel must be rebuilt whenever changes are made to the configuration header.

Note that not all demos, libraries, and tests will build successfully if the prerequisite features are not included.

Kernel options are set by modifying mark3cfg.h, located within the /kernel/public folder.

In the following sections, we will discuss the various configuration options, grouped by functionality.

3.2   Timer Options

KERNEL_USE_TIMERS

This option 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.

Setting this option to 0 disables all timer-based functionality within the kernel.

KERNEL_TIMERS_TICKLESS

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.

Set this option to 1 to use the tickless timer implementation, 0 to use the traditional tick-based approach. Tickless timers are a bit more heavy weight (larger code footprint), but can yield significant power savings as the CPU does not need to wake up at a fixed, high frequency.

KERNEL_USE_TIMEOUTS

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.

Set this option to 1 to enable timeout-based APIs for blocking calls.

KERNEL_USE_QUANTUM

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.

Set this option to 1 to enable round-robin scheduling.

THREAD_QUANTUM_DEFAULT

This value defines the default thread quantum when KERNEL_USE_QUANTUM is enabled. The value defined is a time in milliseconds.

KERNEL_USE_SLEEP

This define enables the Thread::Sleep() API, which allows a thread to suspend its operation for a defined length of time, specified in ms.

3.3 Blocking Objects

KERNEL_USE_NOTIFY

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

KERNEL_USE_SEMAPHORE

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.

Note that all IPC mechanisms (mailboxes, messages) rely on semaphores, so keep in mind that this is a prerequisite for many other features in the kernel.

KERNEL_USE_MUTEX

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.

KERNEL_USE_EVENTFLAG

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.

3.4 Inter-process/thread Communication

KERNEL_USE_MESSAGE

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.

GLOBAL_MESSAGE_POOL_SIZE

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 can share their message objects from this global pool to maximize efficiency and simplify data management.

KERNEL_USE_MAILBOX

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.

3.5 Debug Features

KERNEL_USE_THREADNAME

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.

KERNEL_USE_DEBUG

Provides extra logic for kernel debugging, and instruments the kernel with extra asserts, and kernel trace functionality.

KERNEL_ENABLE_LOGGING

Set this to 1 to enable very chatty kernel logging. Since most important things in the kernel emit logs, a large log-buffer and fast output are required in order to keep up. This is a pretty advanced power-user type feature, so it's disabled by default.

KERNEL_ENABLE_USER_LOGGING

This enables a set of logging macros similar to the kernel-logging macros; however, these can be enabled or disabled independently. This allows for user-code to benefit from the built-in kernel logging macros without having to account for the super-high-volume of logs generated by kernel code.

3.6 Enhancements, Security, Miscellaneous

KERNEL_USE_DRIVER

Enabling device drivers provides a posix-like filesystem interface for peripheral device drivers.

KERNEL_USE_DYNAMIC_THREADS

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.

KERNEL_USE_PROFILER

Provides extra classes for profiling the performance of code. useful for debugging and development, but uses an additional hardware timer.

KERNEL_USE_ATOMIC

Provides support for atomic operations, including addition, subtraction, set, and test-and-set. Add/Sub/Set contain 8, 16, and 32-bit variants.

SAFE_UNLINK

"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.

KERNEL_AWARE_SIMULATION

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.

KERNEL_USE_IDLE_FUNC

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.

KERNEL_USE_AUTO_ALLOC

This feature enables an additional set of APIs that allow for objects to be created on-the-fly out of a special heap, without having to explicitly allocate them (from stack, heap, or static memory). Note that auto-alloc memory cannot be reclaimed.

AUTO_ALLOC_SIZE

Size (in bytes) of the static pool of memory reserved from RAM for use by the auto allocator (if enabled).

Chapter 4

Building Mark3

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.

4.1 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
   export       Platform specific output folder, used when running export.sh
   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
   libs         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 font converter, terminal, programmer, config util
```

4.2 Building the kernel

There are 3 main components of the recursive makefile system used to build Mark3 and its associated middleware libraries and examples. The components are the files "base.mak", "platform.mak", and "build.mak"

The base.mak file determines how the kernel, drivers, and libraries are built, for what targets, and with what options. These options are set as variables that are included in a "platform.mak" file for your target, located under the /builds

directory. "platform.mak" is included for all build steps, and is the place where all chip/board-specific toolchain configuration takes place.

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

Beyond the essential makefiles, the build system uses a series of environment variables to configure a recursive make-based build system appropriately for a given target part and toolchain.

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)
```

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

Once a sane environment has been created, the kernel, libraries, examples and tests can be built by running ./scripts/build.sh from the root directory. By default, Mark3 builds for the atmega328p target, but the target can be selected by manually configuring the above environment variables, or by running the included ./scripts/set_target.sh script as follows:

```
. ./scripts/set_target.sh <architecture> <variant> <toolchain>
```

Where:

```
<architecture> is the target CPU architecture(i.e. avr, msp430, cm0, cm3, cm4f)
<variant>    is the part name (i.e. atmega328p, msp430f2274, generic)
<toolchain>  is the build toolchain (i.e. gcc)
```

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, call the following sequence to guarantee that your code will be built.

```
> make headers
> make library
> make binary
```

Note that library or app-specific environment variables can be set (or modified from the defaults) from within the body of the makefile. For example, the CFLAGS, CPPFLAGS, and LFLAGS variables can be used to supply additional chip- specific toolchain flags. The flags can be used to allow a user to reference chip-specific startup code, headers, middleware, or linker scripts that aren't part of the standard Mark3 distribution.

4.3 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.

Below is an example of setting up the AVR toolchain on Windows:

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 x64, it should look something like this:

```
C:\Program Files (x86)\Atmel\Atmel Toolchain\AVR8 GCC\Native\3.4.2.1002\avr8-gnu-toolchain\bin
```

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.

4.4 Exporting the kernel source

While the build system is flexible enough to adapt to any toolchain, it may be desireable to integrate the Mark3 kernel and associated drivers/libraries into another build system.

Mark3 provides a script (the aptly-named export.sh) which allow for the source for any supported port to be exported for this purpose. This script will also generate appropriate doxygen documentation, and package the whole of it together in a zip file. The files in the archive are placed in a "flat" heirarchy, and do not require any specific path structure to be maintained when imported into another build system.

As a special feature, if the "arduino" AVR target is specified, additional pre-processing is done on the source to turn the standard Mark3 kernel into a library that can be imported directly into Arudino IDE. This is also how the official Mark3 arduino-compatible releases are generated (hosted on mark3os.com and sourceforge.net)

To exercise the build system, type the following from the main mark3 embedded source directory:

```
 > ./scripts/export.sh <target>
```

Where:

Target is one of the following:

```
    atmega328p
    atmega644
    atmega1280
    atmega2560
    atmega1284p
    atxmega256a3
    arduino
    arduino2560
    samd20
    cortex_m0
    cortex_m3
    cortex_m4f
    msp430f2274
```

If successful, the generated artifacats will be placed in an output folder under the ./export directory.

Additionally, if doxygen is found on the host system's PATH, a copy of the manual (using the specific port's source code) will be generated and archived with the source release. If pdflatex is also found on the host's PATH, a PDF copy of the manual will be generated, tailored to the selected target.

Chapter 5

Getting Started With The Mark3 API

5.1 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.

5.2 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.

5.2.1 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.

5.2.2 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.

5.3 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.
}
```

5.4 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();
    }
}
```

5.5 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 artificially 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();
}
```

5.6 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>
    }
}
```

5.7 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.

5.7.1 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.

5.7.2 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.

5.7.3 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 Message↩ Queue 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.

5.7.4 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);
}
```

5.8 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

5.8.1 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.
        ...
    }
}
```

5.9 Notification Objects

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

using this blocking primitive, 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.

5.9.1 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();
}
```

5.10 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;
}
```

5.11 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 6

Why Mark3?

My first job after graduating from university in 2005 was with a small company that had a very old-school, low-budget philosophy when it came to software development.

Every make-or-buy decision ended with "make" when it came to tools. It was the kind of environment where vendors cost us money, but manpower was free. In retrospect, we didn't have a ton of business during the time that I worked there, and that may have had something to do with the fact that we were constantly short on ready cash for things we could code ourselves.

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

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

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

Regardless, I learned a lot about embedded 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 down-loads a month), it was nothing like the success of other RTOS kernels like uC/OS-II and FreeRTOS. To be honest, as a one-man show, I just don't have the resources to support all of the devices, toolchains, and evaluation boards that a real vendor can. I had never expected my kernel to compete with the likes of them, and I don't expect Mark3 to change the embedded landscape either.

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

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

Finally, I wanted to prove that a kernel written entirely in C++ could perform just as well as one written in C. Mark3 is fully benchmarked and profiled – you can see exactly how much it costs to call certain APIs or include various features in the kernel.

And in addition, 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 or handle-passing, type casting, and other operations that are typically considered "unsafe" or "advaned" topics in C.

Chapter 7

When should you use an RTOS?

7.1 The reality of system code

System code can be defined as the program logic required to manage, synchronize, and schedule all of the resources (CPU time, memory, peripherals, etc.) used by the application running on the CPU. And it's true that a significant portion of the code running on an embedded system will be system code. No matter how simple a system is, whether or not this logic is embedded directly into the application (bare-metal system), or included as part of a well-defined stack on which an application is written (RTOS-based); system code is still present, and it comes with a cost.

As an embedded systems is being designed, engineers have to decide which approach to take: Bare-metal, or RTOS. There are advantages and disadvantages to each – and a reasonable engineer should always perform a thorough analysis of the pros and cons of each - in the context of the given application - before choosing a path.

The following figure demonstrates the differences between the architecture of a bare-metal system and RTOS based system at a high level:
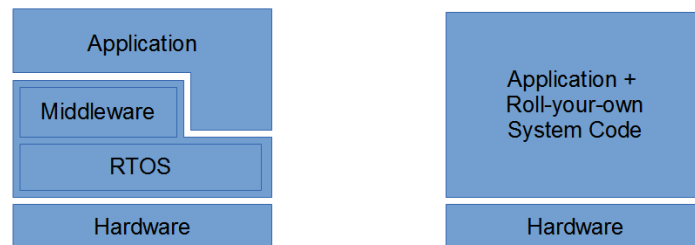
Figure 7.1: Arch

As can be seen, the RTOS (And associated middleware + libraries) captures a certain fixed size.

As a generalization, bare-metal systems typically have the advantage in that the system code overhead is small to start – but grows significantly as the application grows in complexity. At a certain point, it becomes extremely difficult and error-prone to add more functionality to an application running on such a system. There's a tipping point, where the cost of the code used to work-around the limitations of a bare-metal system outweigh the cost of a capable RTOS. Bare-metal systems also generally take longer to implement, because the system code has to be written from scratch (or derived from existing code) for the application. The resulting code also tend to be less portable, as it takes serious discipline to keep the system-specific elements of the code separated – in an RTOS-based system, once the kernel and drivers are ported, the application code is generally platform agnostic.

Conversely, an RTOS-based system incurs a slightly higher fixed cost up-front, but scales infinitely better than a bare-metal system as application's complexity increases. Using an RTOS for simple systems reduces application development time, but may cause an application not to fit into some extremely size-constrained microcontroller. An

RTOS can also cause the size of an application to grow more slowly relative to a bare-metal system – especially as a result of applying synchronization mechanisms and judicious IPC. As a result, an RTOS makes it significantly easier to "go agile" with an application – iteratively adding features and functionality, without having to consider refactoring the underlying system at each turn.

Some of these factors may be more important than others. Requirements, specifications, schedules, chip-selection, and volume projections for a project should all be used to feed into the discussions to decide whether or to go bare-metal or RTOS as a result.

Consider the following questions when making that decision:

- What is the application?

- How efficient is efficient enough?

- How fast is fast enough?

- How small is small enough?

- How responsive is responsive enough?

- How much code space/RAM/etc is available on the target system?

- How much code space/RAM do I need for an RTOS?

- How much code space/RAM do I think I'll need for my application?

- How much time do I have to deliver my system?

- How many units do we plan to sell?

7.2 Superloops, and their limitations

7.2.1 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 bare-metal 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.

7.2.2 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.

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.

7.2.3 Interrupt-Driven Super-loop

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

Fortunately, there is a much more elegant way to do this. In this example, we introduce the concept of the synchronization object. A Synchronization object is some data structure which works within the bounds of the operating system to tell tasks when they can run, and in many cases includes special data unique to the synchronization event.

There are a whole family of synchronization objects, which we'll get into later. In this example, we make use of the simplest synchronization primitive

- the global flag.

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

The following concepts are shown in the example below:

```
volatile K_BOOL something_to_do = false;

__interrupt__ My_Interrupt_Source(void)
{
    something_to_do = true;
}

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

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

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

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

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

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

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

7.2.4 Cooperative multi-tasking

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

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

volatile K_UCHAR event_flags = 0;

// Interrupt sources used to trigger event execution

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

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

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

// Main tasks
int main(void)
{
```

```
    while(1)
    {
        while(event_flags)
        {
            if( event_flags & TASK_1_EVENT)
            {
                Do_Task_1();
                event_flags &= ~TASK_1_EVENT;
            } else if( event_flags & TASK_2_EVENT) {
                Do_Task_2();
                event_flags &= ~TASK_2_EVENT;
            } else if( event_flags & TASK_3_EVENT) {
                Do_Task_3();
                event_flags &= ~TASK_3_EVENT;
            }
        }
        Idle();
    }
}
```

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

Next, we can see that tasks are explicitly given priorities inside the main loop based on the logic of the if/else if structure. As long as there is something set in the event flag, we will always try to execute Task1 first, and only when Task1 isn't set will we attempt to execute Task2, and then Task3. 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 as cooperative multitasking: we can have as many tasks as we want, but unless they cooperate by means of returning back to main, the system can end up with high-priority tasks getting starved for CPU time by lower-priority, long-running tasks.

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

7.2.5 Hybrid cooperative/preemptive multi-tasking

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

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

volatile K_UCHAR hp_tasks = 0;

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

volatile K_UCHAR lp_tasks = 0;

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

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


// Interrupt handler that is used to implement the high-priority event context
__interrupt__ High_Priority_Tasks(void)
{
```

```
    // Enabled every interrupt except this one
    Disable_My_Interrupt();
    Enable_Interrupts();
    while( hp_tasks)
    {
        if( hp_tasks & HP_TASK_1)
        {
            HP_Task1();
            hp_tasks &= ~HP_TASK_1;
        }
        else if (hp_tasks & HP_TASK_2)
        {
            HP_Task2();
            hp_tasks &= ~HP_TASK_2;
        }
    }
    Restore_Interrupts();
    Enable_My_Interrupt();
}

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

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

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

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

7.3   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.

7.3.1   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.

7.3.2   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.

7.3.3   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.

7.3.4   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.

7.3.5   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 8

Can you afford an RTOS?

8.1 Intro

Of course, since you're reading the manual for an RTOS that I've been developing over the course of the several years, you can guess that the conclusion that I draw.

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, the most important being:

Threading, along with priority and time-based scheduling Sophisticated synchronization objects and IPC Flexible, powerful Software Timers Ability to write more portable, decoupled code

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.

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.

8.2 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 realtime I/O functions. As a result, a number of device drivers are also implemented.

Code Space and Memory Overhead:

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 AVR GCC compiler with -Os level optimizations, an executable is produced with the following code/RAM utilization:

```
Program:    27914 bytes
Data:        1313 bytes
```

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:

```
Program:    24886 bytes
Data:         750 bytes
```

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 (ignoring stack use contributions due to interrupts).

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) 128 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 manage the kernel in preemptive mode, and the kernel data itself.

Fixed kernel data costs:

```
--- 134 Bytes Kernel data
--- 26 Bytes Kernel Vtables
```

Application (Variable) data costs:

```
--- 24 Bytes Driver Vtables
--- 123 Bytes - statically-allocated kernel objects (semaphores, timers, etc.)
```

With this taken into account, the true memory cost of a 2-thread system ends up being around 428 bytes of R↩
AM - which is about 20% 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. Also note that by using the "simulated idle" feature provided in
Mark3 R3 and onward, the idle thread (and its associated stack) can be eliminated altogether to reduce the cost in
constrained devices.

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:

```
Kernel ................. 2563 Bytes
Synchronization Objects.  644 Bytes
Port ...................  974 Bytes
Features ...............  871 Bytes
```

The configuration tested in this comparison uses the thread/port module with timers, drivers, and semaphores, and
mutexes, for a total kernel size of 5052 Bytes, with the rest of the code space occupied by the application.

As can be seen from the compiler's output, the difference in code space between the two versions of the application
is 3028 bytes - or about 9% 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. This also demonstrates that the system-software code size in the cooperative
case is about 2024 bytes.

8.3   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 event flag from the timer interrupt

- Acknowledging the event from the event loop

coop – 438 cycles preempt – 764 cycles

Using a cycle-accurate AVR simulator (flAVR) running with a simulated speed of 16MHz, we find the end-to-end event sequence time to be 27us for the cooperative mode scheduler and 48us for the preemptive, and a raw difference of 20us.

With a fixed high-priority event frequency of 30Hz, we achieve a runtime overhead of 611us per second, or 0.06% 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.

8.4 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 25% of a microcontroller's code space/RAM for system code, 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 9

Mark3 Design Goals

9.1 Overview

9.1.1 Services Provided by an RTOS Kernel

At its lowest-levels, an operating system kernel is responsible for managing and scheduling resources within a system according to the application. In a typical thread-based RTOS, the resources involved is CPU time, and the kernel manages this by scheduling threads and timers. But capable RTOS kernels provide much more than just threading and timers.

In the following section, we discuss the Mark3 kernel architecture, all of its features, and a thorough discussion of how the pieces all work together to make an awesome RTOS kernel.

9.1.2 Guiding Principles of Mark3

Mark3 was designed with a number of over-arching principles, coming from years of experience designing, implementing, refining, and experimenting with RTOS kernels. Through that process I not only discovered what features I wanted in an RTOS, but how I wanted to build those features to look, work, and "feel". With that understanding, I started with a clean slate and began designing a new RTOS. Mark3 is the result of that process, and its design goals can be summarized in the following guiding principles.

9.1.3 Be feature competitive

To truly be taken seriously as more than just a toy or educational tool, an RTOS needs to have a certain feature suite. While Mark3 isn't a clone of any existing RTOS, it should at least attempt parity with the most common software in its class.

Looking at its competitors, Mark3 as a kernel supports most, if not all of the compelling features found in modern RTOS kernels, including dynamic threads, tickless timers, efficient message passing, and multiple types of synchronization primitives.

9.1.4 Be highly configuration

Mark3 isn't a one-size-fits-all kernel – and as a result, it provides the means to build a custom kernel to suit your needs. By configuring the kernel at compile-time, Mark3 can be built to contain the optimal feature set for a given application. And since features can be configured individually, you only pay the code/RAM footprint for the features you actually use.

9.1.5 No external dependencies, no new language features

To maximize portability and promote adoption to new platforms, Mark3 is written in a widely supported subset of C++ that lends itself to embedded applications. It avoids RTTI, exceptions, templates, and libraries (C standard, STL, etc.), with all fundamental data structures and types implemented completely for use by the kernel. As a result, the portable parts of Mark3 should compile for any capable C++ toolchain.

9.1.6 Target the most popular hobbyist platforms available

Realistically, this means supporting the various Arduino-compatible target CPUs, including AVR and ARM Cortex-M series microcontrollers. As a result, the current default target for Mark3 is the atmega328p, which has 32KB of flash and 2KB of RAM. All decisions regarding default features, code size, and performance need to take that target system into account.

Mark3 integrates cleanly as a library into the Arduino IDE to support atmega328-based targets. Other AVR and Cortex-M targets can be supported using the port code provided in the source package.

9.1.7 Maximize determinism – but be pragmatic

Guaranteeing deterministic and predictable behavior is tough to do in an embedded system, and often comes with a heavy price tag in either RAM or code-space. With Mark3, we strive to keep the core kernel APIs and features as lightweight as possible, while avoiding algorithms that don't scale to large numbers of threads. We also achieve minimal latency by keeping interrupts enabled (operating out of the critical section) wherever possible.

In Mark3, the most important parts of the kernel are fixed-time, including thread scheduling and context switching. Operations that are not fixed time can be characterized as a function of their dependent data data. For instances, the Mutex and Semaphore APIs operate in fixed time in the uncontested case, and execute in linear time for the contested case – where the speed of execution is dependent on the number of threads currently waiting on that object.

The caveat here is that while we want to minimize latency and time spent in critical sections, that has to be balanced against increases in code size, and uncontested-case performance.

9.1.8 Apply engineering principles – and that means discipline, measurement and verification

My previous RTOS, FunkOS, was designed to be very ad-hoc. The usage instructions were along the lines of "drag and drop the source files into your IDE and compile". There was no regression/unit testing, no code size/speed profiling, and all documentation was done manually. It worked, but the process was a bit of a mess, and resulted in a lot of re-spins of the software, and a lot of time spent stepping through emulators to measure parameters.

We take a different approach in Mark3. Here, we've designed not only the kernel-code, but the build system, unit tests, profiling code, documentation and reporting that supports the kernel. Each release is built and tested using automation in order to ensure quality and correctness, with supporting documentation containing all critical metrics. Only code that passes testing is submitted to the repos and public forums for distribution. These metrics can be traced from build-to-build to ensure that performance remains consistent from one drop to the next, and that no regressions are introduced by new/refactored code.

And while the kernel code can still be exported into an IDE directly, that takes place with the knowledge that the kernel code has already been rigorously tested and profiled. Exporting source in Mark3 is also supported by scripting to ensure reliable, reproducible results without the possibility for human-error.

9.1.9 Use Virtualization For Verification

Mark3 was designed to work with automated simulation tools as the primary means to validate changes to the kernel, due to the power and flexibility of automatic tests on virtual hardware. I was also intrigued by the thought of extending the virtual target to support functionality useful for a kernel, but not found on real hardware.

When the project was started, simavr was the tool of choice- however, its simulation was found to be incorrect compared to execution on a real MCU, and it did not provide the degree of extension that I desired for use with kernel development.

The flAVR AVR simulator was written to replace the dependency on that tool, and overcome those limitations. It also provides a GDB interface, as well as its own built-in debugger, profilers, and trace tools.

The example and test code relies heavily on flAVR kernel aware messaging, so it is recommended that you familiarize yourself with that tool if you intend to do any sort of customizations or extensions to the kernel.

flAVR is hosted on sourceforge at `http://www.sourceforge.net/projects/flavr/` . In its basic configuration, it builds with minimal external dependencies.

- On linux, it requires only pthreads.

- On Windows, it rquires pthreads and ws2_32, both satisfied via MinGW.

- Optional SDL builds for both targets (featuring graphics and simulated joystick input) can be built, and rely on libSDL.

Chapter 10

Mark3 Kernel Architecture

10.1   Overview

At a high level, the Mark3 RTOS is organized into the following features, and layered as shown below:
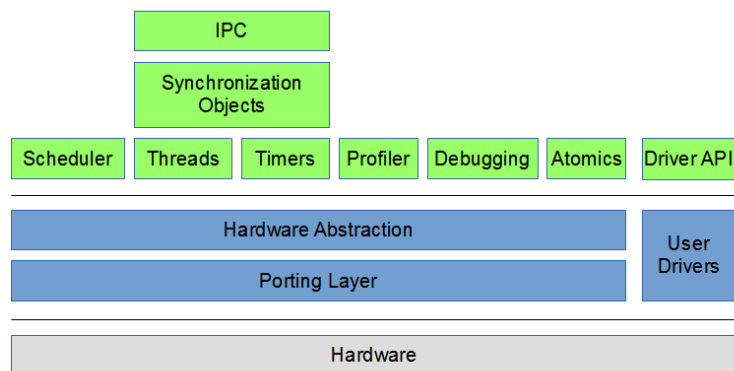
Figure 10.1: Overview

Everything in the "green" layer represents the Mark3 public API and classes, beneath which lives all hardware abstraction and CPU-specific porting and driver code, which runs on a given target CPU.

The features and concepts introduced in this diagram can be described as follows:

Threads: The ability to multiplex the CPU between multiple tasks to give the perception that multiple programs are running simultaneously. Each thread runs in its own context with its own stack.

Scheduler: Algorithm which determines the thread that gets to run on the CPU at any given time. This algorithm takes into account the priorites (and other execution parameters) associated with the threads in the system.

IPC: Inter-process-communications. Message-passing and Mailbox interfaces used to communicate between threads synchronously or asynchronously.

Synchronization Objects: Ability to schedule thread execution relative to system conditions and events, allowing for sharing global data and resources safely and effectively.

Timers: High-resolution software timers that allow for actions to be triggered on a periodic or one-shot basis.

Profiler: Special timer used to measure the performance of arbitrary blocks of code.

Debugging: Realitme logging and trace functionality, facilitating simplified debugging of systems using the OS.

Atomics: Support for UN-interruptble arithmatic operations.

Driver API: Hardware abstraction interface allowing for device drivers to be written in a consistent, portable manner.

Hardware Abstraction Layer: Class interface definitions to represent threading, context-switching, and timers in a generic, abstracted manner.

Porting Layer: Class interface implementation to support threading, context-switching, and timers for a given CPU.

User Drivers: Code written by the user to implement device-specific peripheral drivers, built to make use of the Mark3 driver API.

Each of these features will be described in more detail in the following sections of this chapter.

The concepts introduced in the above architecture are implemented in a variety of source modules, which are logically broken down into classes (or in some cases, groups of functions/macros). The relationship between objects in the Mark3 kernel is shown below:
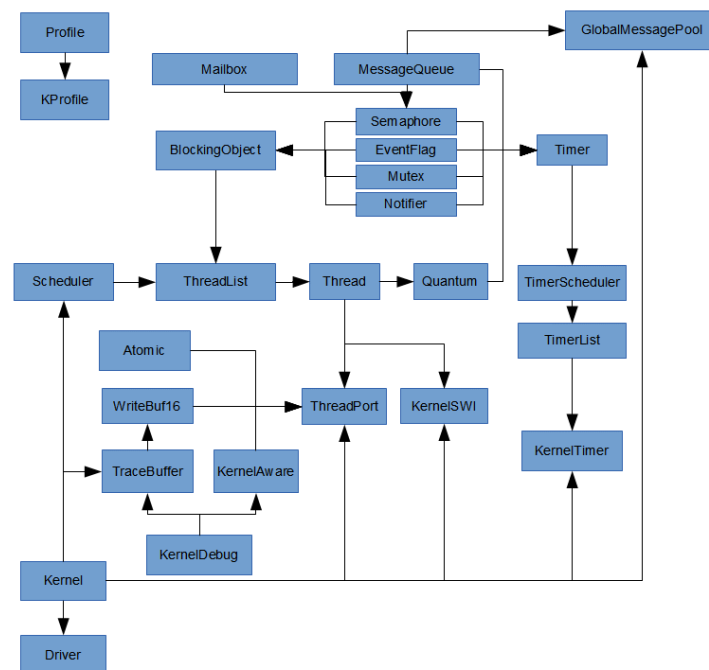
Figure 10.2: Overview

The objects shown in the preceding table can be grouped together by feature. In the table below, we group each feature by object, referencing the source module in which they can be found in the Mark3 source tree.

Feature	**Kernel** Object	Source Files
Profiling	ProfileTimer	profile.cpp/.h
Threads + Scheduling	Thread	thread.cpp/.h
	Scheduler	scheduler.cpp/.h
	PriorityMap	priomap.cpp/.h
	Quantum	quantum.cpp/.h
	ThreadPort	threadport.cpp/.h ∗∗
	KernelSWI	kernelswi.cpp/.h ∗∗
Timers	Timer	timer.h/timer.cpp
	TimerScheduler	timerscheduler.h
	TimerList	timerlist.h/cpp

	KernelTimer	kerneltimer.cpp/.h **
Synchronization	BlockingObject	blocking.cpp/.h
	Semaphore	ksemaphore.cpp/.h
	EventFlag	eventflag.cpp/.h
	Mutex	mutex.cpp/.h
	Notify	notify.cpp/.h
IPC/Message-passing	Mailbox	mailbox.cpp/.h
	MessageQueue	message.cpp/.h
	GlobalMessagePool	message.cpp/.h
Debugging	Miscellaneous Macros	kerneldebug.h
	KernelAware	kernelaware.cpp/.h
	TraceBuffer	tracebuffer.cpp/.h
	Buffalogger	buffalogger.h
Device Drivers	Driver	driver.cpp/.h
Atomic Operations	Atomic	atomic.cpp/.h
Kernel	Kernel	kernel.cpp/.h

```
** implementation is platform-dependent, and located under the kernel's
** /cpu/<arch>/<variant>/<toolchain> folder in the source tree
```

10.2   Threads and Scheduling

The classes involved in threading and scheudling in Mark3 are highlighted in the following diagram, and are discussed in detail in this chapter:
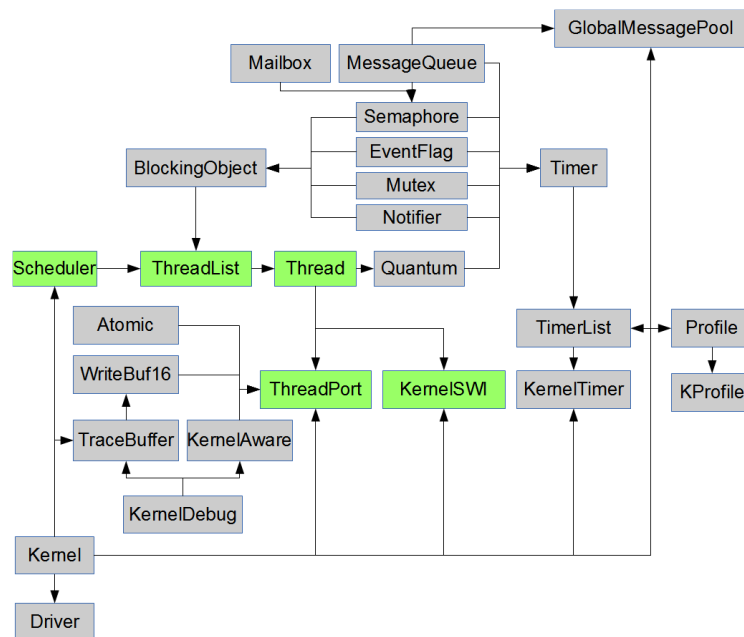
Figure 10.3: Threads and Scheduling

10.2.1   A Bit About Threads

Before we get started talking about the internals of the Mark3 scheduler, it's necessary to go over some background material - starting with: what is a thread, anyway?

Let's look at a very basic CPU without any sort of special multi-threading hardware, and without interrupts. When the CPU is powered up, the program counter is loaded with some default location, at which point the processor core will start executing instructions sequentially - running forever and ever according to whatever has been loaded into program memory. This single instance of a simple program sequence is the only thing that runs on the processor, and the execution of the program can be predicted entirely by looking at the CPU's current register state, its program, and any affected system memory (the CPU's "context").

It's simple enough, and that's exactly the definition we have for a thread in an RTOS.

Each thread contains an instance of a CPU's register context, its own stack, and any other bookkeeping information necessary to define the minimum unique execution state of a system at runtime. It is the job of a RTOS to multiplex the execution of multiple threads on a single physical CPU, thereby creating the illusion that many programs are being executed simultaneously. In reality there can only ever be one thread truly executing at any given moment on a CPU core, so it's up to the scheduler to set and enforce rules about what thread gets to run when, for how long, and under what conditions. As mentioned earlier, any system without an RTOS exeuctes as a single thread, so at least two threads are required for an RTOS to serve any useful purpose.

Note that all of this information is is common to pretty well every RTOS in existence - the implementation details, including the scheduler rules, are all part of what differentiates one RTOS from another.

10.2.2 Thread States and ThreadLists

Since only one thread can run on a CPU at a time, the scheduler relies on thread information to make its decisions. Mark3's scheduler relies on a variety of such information, including:

- The thread's current priority

- Round-Robin execution quanta

- Whether or not the thread is blocked on a synchronization object, such as a mutex or semaphore

- Whether or not the thread is currently suspended

The scheduler further uses this information to logically place each thread into 1 of 4 possible states:

```
- Ready - The thread is currently running
- Running - The thread is able to run
- Blocked - The thread cannot run until a system condition is met
- Stopped - The thread cannot run because its execution has been suspended
.
```

In order to determine a thread's state, threads are placed in "buckets" corresponding to these states. Ready and running threads exist in the scheduler's buckets, blocked threads exist in a bucket belonging to the object they're blocked on, and stopped threads exist in a separate bucket containing all stopped threads.

In reality, the various buckets are just doubly-linked lists of Thread objects - implemented in something called the ThreadList class. To facilitate this, the Thread class directly inherits from a LinkListNode class, which contains the node pointers required to implement a doubly-linked list. As a result, Threads may be effortlessly moved from one state to another using efficient linked-list operations built into the ThreadList class.

10.2.3 Blocking and Unblocking

While many developers new to the concept of an RTOS assume that all threads in a system are entirely separate from eachother, the reality is that practical systems typically involve multiple threads working together, or at the very least sharing resources. In order to synchronize the execution of threads for that purpose, a number of synchronization primitives (blocking objects) are implemented to create specific sets of conditions under which threads can continue execution. The concept of "blocking" a thread until a specific condition is met is fundamental to understanding RTOS applications design, as well as any highly-multithreaded applications.

10.2.4 Blocking Objects

Blocking objects and primitives provided by Mark3 include:

- Semaphores (binary and counting)

- Mutexes

- Event Flags

- Thread Notification Objects

- Thread Sleep

- Message Queues

- Mailboxes

The relationship between these objects in the system are shown below:
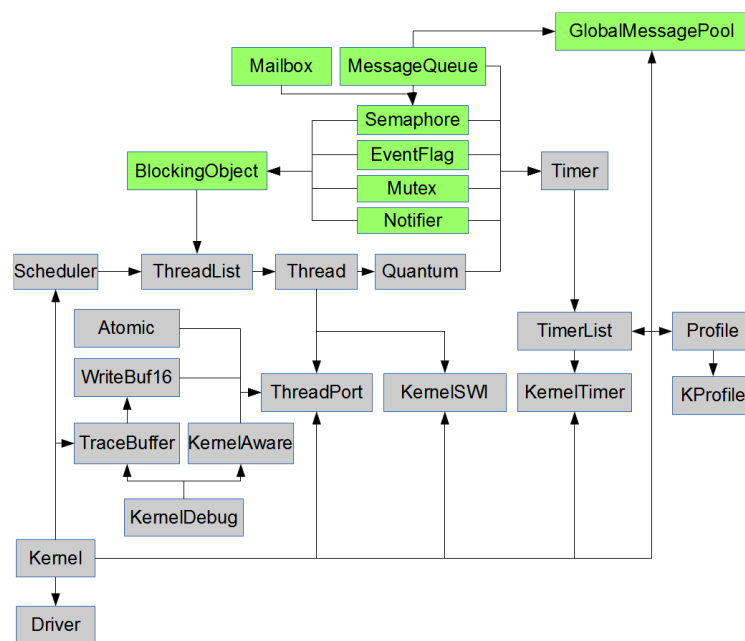
Figure 10.4: Blocking Objects

Each of these objects inherit from the BlockingObject class, which itself contains a ThreadList object. This class contains methods to Block() a thread (remove it from the Scheduler's "Ready" or "Running" ThreadLists), as well as UnBlock() a thread (move a thread back to the "Ready" lists). This object handles transitioning threads from list-to-list (and state-to-state), as well as taking care of any other Scheduler bookkeeping required in the process. While each of the Blocking types implement a different condition, they are effectively variations on the same theme. Many simple Blocking objects are also used to build complex blocking objects - for instance, the Thread Sleep mechanism is essentially a binary semaphore and a timer object, while a message queue is a linked-list of message objects combined with a semaphore.

10.3 Inside the Mark3 Scheduler

At this point we've covered the following concepts:

- Threads

- Thread States and Thread Lists

- Blocking and Un-Blocking Threads

Thankfully, this is all the background required to understand how the Mark3 Scheduler works. In technical terms, Mark3 implements "strict priority scheduling, with round-robin scheduling among threads in each priority group". In plain English, this boils down to a scheduler which follows a few simple rules:

```
Find the highest-priority "Ready" list that has at least one Threads.
If the first thread in that bucket is not the current thread, select it
to run next. Otherwise, rotate the linked list, and choose the next
thread in the list to run
```

Since context switching is one of the most common and frequent operation performed by an RTOS, this needs to be as fast and deterministic as possible. While the logic is simple, a lot of care must be put into optimizing the scheduler to achieve those goals. In the section below we discuss the optimization approaches taken in Mark3.

There are a number of ways to find the highest-priority thread. The naive approach would be to simply iterate through the scheduler's array of ThreadLists from highest to lowest, stopping when the first non-empty list is found, such as in the following block of code:

```
for (prio = num_prio - 1; prio >= 0; prio--)
{
    if (thread_list[prio].get_head() != NULL)
    {
        break;
    }
}
```

While that would certainly work and be sufficient for a variety of systems, it's a non-deterministic approach (complexity O(n)) whose cost varies substantially based on how many priorities have to be evaluated. It's simple to read and understand, but it's non-optimal.

Fortunatley, a functionally-equivalent and more deterministic approach can be implemented with a few tricks.

In addition to maintaining an array of ThreadLists, Mark3 also maintains a bitmap (one bit per priority level) that indicates which thread lists have ready threads. This bitmap is maintained automatically by the ThreadList class, and is updated every time a thread is moved to/from the Scheduler's ready lists.

By inspecting this bitmap using a technique to count the leading zero bits in the bitmap, we determine which threadlist to choose in fixed time.

Now, to implement the leading-zeros check, this can once again be performed iteratively using bitshifts and compares (which isn't any more efficient than the raw list traversal), but it can also be evaluated using either a lookup table, or via a special CPU instruction to count the leading zeros in a value. In Mark3, we opt for the lookup-table approach since we have a limited number of priorities and not all supported CPU architectures support a count leading zero instruction. To achieve a balance between performance and memory use, we use a 4-bit lookup table (costing 16 bytes) to perform the lookup.

(As a sidenote - this is actually a very common approach in OS schedulers. It's actually part of the reason why modern ARM cores implement a dedicated count-leading-zeros [CLZ] instruction!)

With a 4-bit lookup table and an 8-bit priority-level bitmap, the priority check algorithm looks something like this:

```
// Check the highest 4 priority levels, represented in the
// upper 4 bits in the bitmap
priority = priority_lookup_table[(priority_bitmap >> 4)];

// priority is non-zero if we found something there
if( priority )
```

```
{
    // Add 4 because we were looking at the higher levels
    priority += 4;
}
else
{
    // Nothing in the upper 4, look at the lowest 4 priority levels
    // represented by the lowest 4 bits in the bitmap
    priority = priority_lookup_table[priority_bitmap & 0x0F];
}
```

Deconstructing this algorithm, you can see that the priority lookup will have on O(1) complexity - and is extremely low-cost.

This operation is thus fully deterministic and time bound - no matter how many threads are scheduled, the operation will always be time-bound to the most expensive of these two code paths. Even with only 8 priority levels, this is still much faster than iteratively checking the thread lists manually, compared with the previous example implementation.

Once the priority level has been found, selecting the next thread to run is trivial, consisting of something like this:

next_thread = thread_list[prio].get_head();

In the case of the get_head() calls, this evaluates to an inline-load of the "head" pointer in the particular thread list.

One important thing to take away from this analysis is that the scheduler is only responsible for selecting the next-to-run thread. In fact, these two operations are totally decoupled - no context switching is performed by the scheduler, and the scheduler isn't called from the context switch. The scheduler simply produces new "next thread" values that are consumed from within the context switch code.

10.3.1 Considerations for Round-Robin Scheduling

One thing that isn't considered directly from the scheduler algorithm is the problem of dealing with multiple threads within a single priority group; all of the alorithms that have been explored above simply look at the first Thread in each group.

Mark3 addresses this issue indirectly, using a software timer to manage round-robin scheduling, as follows.

In some instances where the scheduler is run by the kernel directly (typically as a result of calling Thread::Yield()), the kernel will perfom an additional check after running the Scheduler to determine whether or there are multiple ready Threadsin the priority of the next ready thread.

If there are multiple threads within that priority, the kernel adds a one-shot software timer which is programmed to expire at the next Thread's configured quantum. When this timer expires, the timer's callback function executes to perform two simple operations:

"Pivot" the current Thread's priority list. Set a flag telling the kernel to trigger a Yield after exiting the main Timer↩ Scheduler processing loop

Pivoting the thread list basically moves the head of a circular-linked-list to its next value, which in our case ensures that a new thread will be chosen the next time the scheduler is run (the scheduler only looks at the head node of the priority lists). And by calling Yield, the system forces the scheduler t run, a new round-robin software timer to be installed (if necssary), and triggers a context switch SWI to load the newly-chosen thread. Note that if the thread attached to the round-robin timer is pre-empted, the kernel will take steps to abort and invalidate that round-robin software timer, installing a new one tied to the next thread to run if necessary.

Because the round-robin software timer is dynamically installed when there are multiple ready threads at the highest ready priority level, there is no CPU overhead with this feature unless that condition is met. The cost of round-robin scheduling is also fixed - no matter how many threads there are, and the cost is identical to any other one-shot software timer in the system.

10.3.2 Context Switching

There's really not much to say about the actual context switch operation at a high level. Context switches are triggered whenever it has been determined that a new thread needs to be swapped into the CPU core when the scheduler is run. Mark3 implements also context switches as a call to a software interrupt - on AVR platforms, we

typically use INT0 or INT2 for this (although any pin-change GPIO interrupt can be used), and on ARM we achieve this by triggering a PendSV exception.

However, regardless of the architecture, the contex-switch ISR will perform the following three operations:

Save the current Thread's context to the current Thread stack Make the "next to run" thread the "currently running" thread Restore the context of the next Thread from the Thread stack

The code to implement the context switch is entirely architecture-specific, so it won't be discussed in detail here. It's almost always gory inline-assembly which is used to load and store various CPU registers, and is highly-optimized for speed. We dive into an example implementation for the ARM Cortex-M0 microcontroller in a later section of this book.

10.3.3   Putting It All Together

In short, we can say that the Mark3 scheduler works as follows:

- The scheduler is run whenever a Thread::Yield() is called by a user, as part of blocking calls, or whenever a new thread is started

- The Mark3 scheduler is deterministic, selecting the next thread to run in fixed-time

- The scheduler only chooses the next thread to run, the context switch SWI consumes that information to get that thread running

- Where there are multiple ready threads in the highest populated priority level, a software timer is used to manage round-robin scheduling

While we've covered a lot of ground in this section, there's not a whole lot of code involved. However, the code that performs these operations is nuanced and subtle. If you're interested in seeing how this all works in practice, I suggest reading through the Mark3 source code (which is heavily annotated), and stepping through the code with a simulator/emulator.

10.4   Timers

Mark3 implements one-shot and periodic software-timers via the Timer class. The user configures the timer for duration, repetition, and action, at which point the timer can be activated. When an active timer expires, the kernel calls a user-specified callback function, and then reloads the timer in the case of periodic timers. The same timer objects exposed to the user are also used within the kernel to implement round-robin scheduling, and timeout-based APIs for seamphores, mutexes, events, and messages.

Timers are implemented using the following components in the Mark3 Kernel:

Figure 10.5: Timers

The Timer class provides the basic periodic and one-shot timer functionality used by applicaiton code, blocking objects, and IPC.

The TimerList class implements a doubly-linked list of Timer objects, and the logic required to implement a timer tick (tick-based kernel) or timer expiry (tickless kernel) event.

The TimerScheduler class contains a single TimerList object, implementing a single, system-wide list of Timer objects within the kernel. It also provides hooks for the hardware timer, such that when a timer tick or expiry event occurs, the TimerList expiry handler is run.

The KernelTimer class (kerneltimer.cpp/.h) implements the CPU specific hardware timer driver that is used by the kernel and the TimerScheduler to implement software timers.

While extremely simple to use, they provide one of the most powerful execution contexts in the system.

The software timers implemented in Mark3 use interrupt-nesting within the kernel timer's interrupt handler. This context is be considered higher-priority than the highest priority user thread, but lower-priority than other interrupts in the system. As a result, this minimizes critical interrupt latency in the system, albeit at the expense of responsiveness of the user-threads.

For this reason, it's critical to ensure that all timer callback events are kept as short as possible to prevent adding thread-level latency. All heavy-lifting should be left to the threads, so the callback should only implement signalling via IPC or synchronization object.

The time spent in this interrupt context is also dependent on the number of active timers at any given time. However, Mark3 also can be used to minimize the frequency of these interrupts wakeups, by using an optional "tolerance" parameter in the timer API calls. In this way, periodic tasks that have less rigorous real-time constraints can all be grouped together – executing as a group instead of one-after-another.

Mark3 also contains two different timer implementations that can be configured at build-time, each with their own advantages.

10.4.1 Tick-based Timers

In a tick-based timing scheme, the kernel relies on a system-timer interrupt to fire at a relatively-high frequency, on which all kernel timer events are derived. On modern CPUs and microcontrollers, a 1kHz system tick is common, although quite often lower frequencies such as 60Hz, 100Hz, or 120Hz are used. The resolution of this timer also defines the maximum resolution of timer objects as a result. That is, if the timer frequency is 1kHz, a user cannot specify a timer resolution lowerthan 1ms.

The advantage of a tick-based timer is its sheer simplicity. It typically doesn't take much to set up a timer to trigger an interrupt at a fixed-interval, at which point, all system timer intervals are decremented by 1 count. When each system timer interval reaches zero, a callback is called for the event, and the events are either reset and restarted (repeated timers) or cleared (1-shot).

Unfortunately, that simplicity comes at a cost of increased interrupt count, which cause frequent CPU wakeups and utilization, and power consumption.

10.4.2 Tickless Timers

In a tickless system, the kernel timer only runs when there are active timers pending expiry, and even then, the timer module only generates interrupts when a timer expires, or a timer reaches its maximum count value. Additionally, when there are no active timer objects, the timer can is completely disabled – saving even more cycles, power, and CPU wakeups. These factors make the tickless timer approach a highly-optimal solution, suitable for a wide array of low-power applications.

Also, since tickless timers do not rely on a fixed, periodic clock, they can potentially be higher resolution. The only limitation in timer resolution is the precision of the underlying hardware timer as configured. For example, if a 32kHz hardware timer is being used to drive the timer scheduler, the resolution of timer objects would be in the ~33us range.

The only downside of the tickless timer system is an added complexity to the timer code, requiring more code space, and slightly longer execution of the timer routines when the timer interrupt is executed.

10.4.3 Timer Processing Algorithm

Timer interrupts occur at either a fixed-frequency (tick-based), or at the next timer expiry interval (tickless), at which point the timer processing algorithm runs. While the timer count is reset by the timer-interrupt, it is still allowed to accumulate ticks while this algorithm is executed in order to ensure that timer-accuracy is kept in real-time. It is also important to note that round-robin scheduling changes are disabled during the execution of this algorithm to prevent race conditions, as the round-robin code also relies on timer objects.

All active timer objects are stored in a doubly-linked list within the timer-scheduler, and this list is processed in two passes by the alogirthm which runs from the timer-interrupt (with interrupt nesting enabled). The first pass determines which timers have expired and the next timer interval, while the second pass deals with executing the timer callbacks themselves. Both phases are discussed in more detail below.

In the first pass, the active timers are decremented by either 1 tick (tick-based), or by the duration of the last elapsed timer interval (tickless). Timers that have zero (or less-than-zero) time remaining have a "callback" flag set, telling the algorithm to call the timer's callback function in the second pass of the loop. In the event of a periodic timer, the timer's interval is reset to its starting value.

For the tickless case, the next timer interval is also computed in the first-pass by looking for the active timer with the least amount of time remaining in its interval. Note that this calculation is irrelevant in the tick-based timer code, as the timer interrupt fires at a fixed-frequency.

In the second pass, the algorithms loops through the list of active timers, looking for those with their "callback" flag set in the first pass. The callback function is then executed for each expired timer, and the "callback" flag cleared. In the event that a non-periodic (one-shot) timer expires, the timer is also removed from the timer scheduler at this time.

In a tickless system, once the second pass of the loop has been completed, the hardware timer is checked to see if the next timer interval has expired while processing the expired timer callbacks. In that event, the complete

algorithm is re-run to ensure that no expired timers are missed. Once the algorithm has completed without the next timer expiring during processing, the expiry time is programmed into the hardware timer. Round-robin scheduling is re-enabled, and if a new thread has been scheduled as a result of action taken during a timer callback, a context switch takes place on return from the timer interrupt.

10.5 Synchronization and IPC
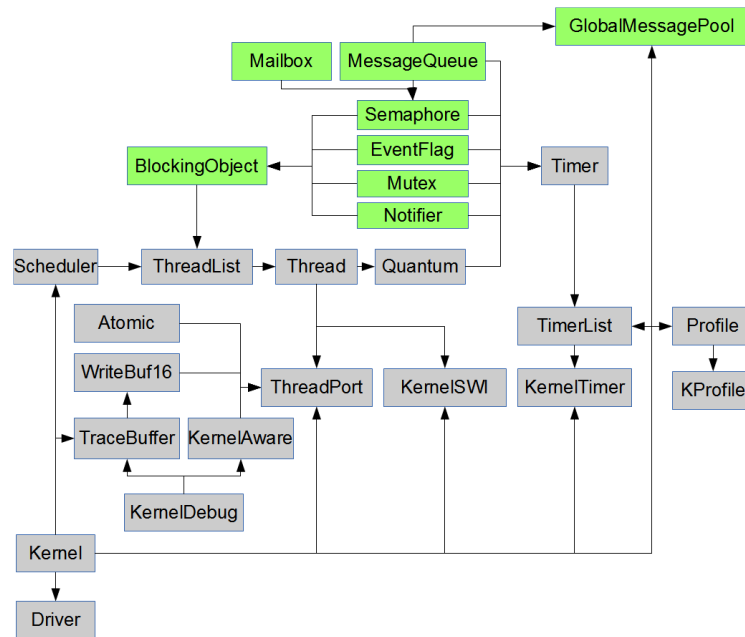
Figure 10.6: Synchronization and IPC

10.6 Blocking Objects

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

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

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

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

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

Mark3 implements a variety of blocking objects including semaphores, mutexes, event flags, and IPC mechanisms that all inherit from the basic Blocking-object class found in blocking.h/cpp, ensuring consistency and a high degree of code-reuse between components.

10.6.1 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. Semaphores can also be posted (but not pended) from within the interrupt context.

10.6.2 Mutex

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 recursive in Mark3 - that is, the owner thread can claim a mutex more than once. The caveat here is that a recursively-held mutex will not be released until a matching "release" call is made for each "claim" call.

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.

10.6.3 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.

10.6.4 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 notification has been signalled, all threads currently blocked on the object become unblocked and moved into the ready list.

Signalling a notification object that has no actively-waiting threads has no effect.

10.7 Messages and Global Message Queue

10.7.1 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.

10.7.2 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.

10.7.3 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 will occur, which may cripple the operating system's ability to pass data between threads.

10.7.4 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 Message↩ Queue 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.

10.7.5 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

10.7.6 Atomic Operations
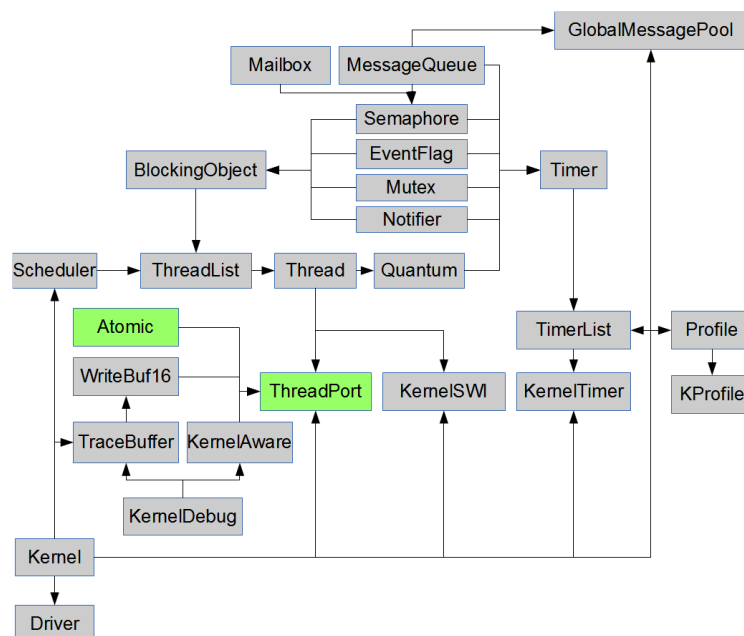
Figure 10.7: Atomic operations

This utility class provides primitives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primitives provided here include Set/Add/Delete for 8, 16, and 32-bit integer types, as well as an atomic test-and-set.

10.7.7   Drivers
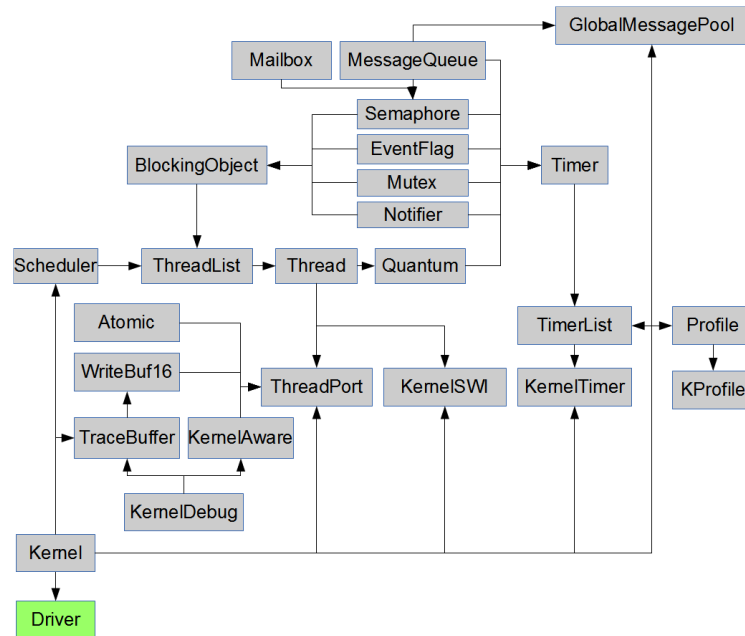
Figure 10.8: Drivers

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.

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.

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 argument - the pointer to the 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!");
```

10.8 Kernel Proper and Porting
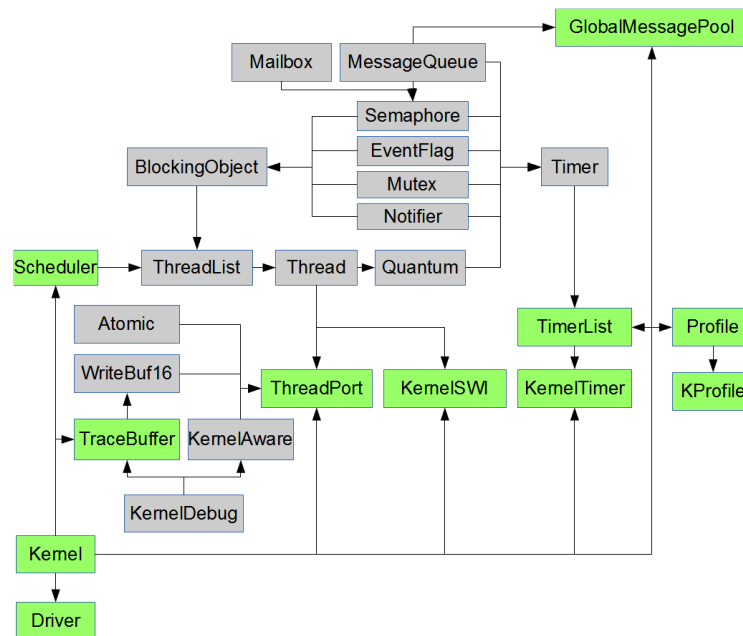
Figure 10.9: Kernel Proper and Porting

The Kernel class is a static class with methods to handle the initialization and startup of the RTOS, manage errors, and provide user-hooks for fatal error handling (functions called when Kernel::Panic() conditions are encountered), or when the Idle function is run.

Internally, Kernel::Init() calls the initialization routines for various kernel objects, providing a single interface by which all RTOS-related system initialization takes place.

Kernel::Start() is called to begin running OS funcitonality, and does not return. Control of the CPU is handed over to the scheduler, and the highest-priority ready thread begins execution in the RTOS environment.

Harware Abstraction Layer

Almost all of the Mark3 kernel (and middleware) is completely platform independent, and should compile cleanly on any platform with a modern C++ compiler. However, there are a few areas within Mark3 that can only be implemented by touching hardware directly.

These interfaces generally cover four features:

- Thread initializaiton and context-switching logic

- Software interrupt control (used to generate context switches)

- Hardware timer control (support for time-based functionlity, such as Sleep())

- Code-execution profiling timer (not necessary to port if code-profiling is not compiled into the kernel)

The hardware abstraction layer in Mark3 provides a consistent interface for each of these four features. Mark3 is ported to new target architectures by providing an implementation for all of the interfaces declared in the abstraction layer. In the following section, we will explore how this was used to port the kernel to ARM Cortex-M0.

Real-world Porting Example – Cortex M0

This section serves as a real-world example of how Mark3 can be ported to new architectures, how the Mark3 abstraction layer works, and as a practical reference for using the RTOS support functionality baked in modern A←RM Cortex-M series microcontrollers. Most of this documentation here is taken directly from the source code found in the /kernel/cpu/cm0/ ports directory, with additional annotations to explain the port in more detail. Note that a familiarity with Cortex-M series parts will go a long way to understanding the subject matter presented, especially a basic understanding of the ARM CPU registers, exception models, and OS support features (PendSV, SysTick and SVC). If you're unfamiliar with ARM architecture, pay attention to the comments more than the source itself to illustrate the concepts.

Porting Mark3 to a new architecture consists of a few basic pieces; for developers familiar with the target architecture and the porting process, it's not a tremendously onerous endeavour to get Mark3 up-and-running somewhere new. For starters, all non-portable components are completely isolated in the source-tree under:

/embedded/kernel/CPU/VARIANT/TOOLCHAIN/,

where CPU is the architecture, VARIANT is the vendor/part, and TOOLCHAIN is the compiler tool suite used to build the code.

From within the specific port folder, a developer needs only implement a few classes and headers that define the port-specific behavior of Mark3:

- KernelSWI (kernelswi.cpp/kernelswi.h) - Provides a maskable software-triggered interrupt used to perform context switching.

- KernelTimer (kerneltimer.cpp/kerneltimer.h) - Provides either a fixed-frequency or programmable-interval timer, which triggers an interrupt on expiry. This is used for implementing round-robin scheduling, thread-sleeps, and generic software timers.

- Profiler (kprofile.cpp/kprofile.h) - Contains code for runtime code-profiling. This is optional and may be stubbed out if left unimplemented (we won't cover profiling timers here).

- ThreadPort (threadport.cpp/threadport.h) - The meat-and-potatoes of the port code lives here. This class contains architecture/part-specific code used to initialize threads, implement critical-sections, perform context-switching, and start the kernel. Most of the time spent in this article focuses on the code found here.

Summarizing the above, these modules provide the following list of functionality:

```
- Thread stack initialization
- Kernel startup and first thread entry
- Context switch and SWI
- Kernel timers
- Critical Sections
.
```

The implementation of each of these pieces will be analyzed in detail in the sections that follow.

Thread Stack Initialization

Before a thread can be used, its stack must first be initialized to its default state. This default state ensures that when the thread is scheduled for the first time and its context restored, that it will cause the CPU to jump to the user's specified entry-point function.

All of the platform independent thread setup is handled by the generic kernel code. However, since every CPU architecture has its own register set, and stacks different information as part of an interrupt/exception, we have to implement this thread setup code for each platform we want the kernel to support (Combination of Architecture + Variant + Toolchain).

In the ARM Cortex-M0 architecture, the stack frame consists of the following information:

a) Exception Stack Frame

Contains the 8 registers which the ARM Cortex-M0 CPU automatically pushes to the stack when entering an exception. The following registers are included (in stack'd order):

```
[ XPSR ] <-- Highest address in context
[ PC   ]
```

```
[ LR   ]
[ R12  ]
[ R3   ]
[ R2   ]
[ R1   ]
[ R0   ]
```

XPSR – This is the CPU's status register. We need to set this to 0x01000000 (the "T" bit), which indicates that the CPU is executing in "thumb" mode. Note that ARMv6m and ARMv7m processors only run thumb2 instructions, so an exception is liable to occur if this bit ever gets cleared.

PC – Program Counter. This should be set with the initial entry point (function pointer) for that the user wishes to start executing with this thread.

LR - The base link register. Normally, this register contains the return address of the calling function, which is where the CPU jumps when a function returns. However, our threads generally don't return (and if they do, they're placed into the stop state). As a result we can leave this as 0.

The other registers in the stack frame are generic working registers, and have no special meaning, with the exception that R0 will hold the user's argument value passed into the entrypoint.

b) Complimentary CPU Register Context

```
[ R11  ]
...
[ R4   ] <-- Lowest address in context
```

These are the other general-purpose CPU registers that need to be backed up/ restored on a context switch, but aren't stacked by default on a Cortex-M0 exception. If there were any additional hardware registers to back up, then we'd also have to include them in this part of the context as well.

As a result, these registers all need to be manually pushed to the stack on stack creation, and will need to be explicitly pushed and pop as part of a normal context switch.

With this default exception state in mind, the following code is used to initialize a thread's stack for a Cortex-M0.

```cpp
void ThreadPort::InitStack(Thread *pclThread_)
{
    K_ULONG *pulStack;
    K_ULONG *pulTemp;
    K_ULONG ulAddr;
    K_USHORT i;

    // Get the entrypoint for the thread
    ulAddr = (K_ULONG)(pclThread_->m_pfEntryPoint);

    // Get the top-of-stack pointer for the thread
    pulStack = (K_ULONG*)pclThread_->m_pwStackTop;

    // Initialize the stack to all FF's to aid in stack depth checking
    pulTemp = (K_ULONG*)pclThread_->m_pwStack;
    for (i = 0; i < pclThread_->m_usStackSize / sizeof(K_ULONG); i++)
    {
        pulTemp[i] = 0xFFFFFFFF;
    }

    PUSH_TO_STACK(pulStack, 0);                 // Apply one word of padding

    //-- Simulated Exception Stack Frame --
    PUSH_TO_STACK(pulStack, 0x01000000);     // XSPR;set "T" bit for thumb-mode
    PUSH_TO_STACK(pulStack, ulAddr);         // PC
    PUSH_TO_STACK(pulStack, 0);              // LR
    PUSH_TO_STACK(pulStack, 0x12);
    PUSH_TO_STACK(pulStack, 0x3);
    PUSH_TO_STACK(pulStack, 0x2);
    PUSH_TO_STACK(pulStack, 0x1);
    PUSH_TO_STACK(pulStack, (K_ULONG)pclThread_->m_pvArg);     // R0 = argument

    //-- Simulated Manually-Stacked Registers --
    PUSH_TO_STACK(pulStack, 0x11);
    PUSH_TO_STACK(pulStack, 0x10);
    PUSH_TO_STACK(pulStack, 0x09);
    PUSH_TO_STACK(pulStack, 0x08);
    PUSH_TO_STACK(pulStack, 0x07);
    PUSH_TO_STACK(pulStack, 0x06);
    PUSH_TO_STACK(pulStack, 0x05);
```

```
    PUSH_TO_STACK(pulStack, 0x04);
    pulStack++;

    pclThread_->m_pwStackTop = pulStack;
}
```

Kernel Startup

The same general process applies to starting the kernel on an ARM Cortex-M0 as on other platforms. Here, we initialize and start the platform specific timer and software-interrupt modules, find the first thread to run, and then jump to that first thread.

Now, to perform that last step, we have two options:

1) Simulate a return from an exception manually to start the first thread, or.. 2) Use a software interrupt to trigger the first "Context Restore/Return from Interrupt"

For 1), we basically have to restore the whole stack manually, not relying on the CPU to do any of this for us. That's certainly doable, but not all Cortex parts support this (other members of the family support privileged modes, etc.). That, and the code required to do this is generally more complex due to all of the exception-state simulation. So, we will opt for the second option instead.

To implement a software to start our first thread, we will use the SVC instruction to generate an exception. From that exception, we can then restore the context from our first thread, set the CPU up to use the right "process" stack, and return-from-exception back to our first thread. We'll explore the code for that later.

But, before we can call the SVC exception, we're going to do a couple of things.

First, we're going to reset the default MSP stack pointer to its original top-of-stack value. The rationale here is that we no longer care about the data on the MSP stack, since calling the SVC instruction triggers a chain of events from which we never return. The MSP is also used by all exception-handling, so regaining a few words of stack here can be useful. We'll also enable all maskable exceptions at this point, since this code results in the kernel being started with the CPU executing the RTOS threads, at which point a user would expect interrupts to be enabled.

Note, the default stack pointer location is stored at address 0x00000000 on all ARM Cortex M0 parts. That explains the code below...

```
void ThreadPort_StartFirstThread( void )
{
    asm(
        " ldr r1, [r0] \n" // Reset the MSP to the default base address
        " msr msp, r1 \n"
        " cpsie i \n"      // Enable interrupts
        " svc 0 \n"        // Jump to SVC Call
        );
}
```

First Thread Entry

This handler has the job of taking the first thread object's stack, and restoring the default state data in a way that ensures that the thread starts executing when returning from the call.

We also keep in mind that there's an 8-byte offset from the beginning of the thread object to the location of the thread stack pointer. This offset is a result of the thread object inheriting from the linked-list node class, which has 8-bytes of data. This is stored first in the object, before the first element of the class, which is the "stack top" pointer.

The following assembly code shows how the SVC call is implemented in Mark3 for the purpose of starting the first thread.

```
get_thread_stack:
    ; Get the stack pointer for the current thread
    ldr r0, g_pstCurrent
    ldr r1, [r0]
    add r1, #8
    ldr r2, [r1]          ; r2 contains the current stack-top

load_manually_placed_context_r11_r8:
    ; Handle the bottom 32-bytes of the stack frame
    ; Start with r11-r8, because only r0-r7 can be used
    ; with ldmia on CM0.
    add r2, #16
    ldmia r2!, {r4-r7}
```

```
    mov r11, r7
    mov r10, r6
    mov r9, r5
    mov r8, r4

set_psp:
    ; Since r2 is coincidentally back to where the stack pointer should be,
    ; Set the program stack pointer such that returning from the exception handler
    msr psp, r2

load_manually_placed_context_r7_r4:
    ; Get back to the bottom of the manually stacked registers and pop.
    sub r2, #32
    ldmia r2!, {r4-r7}  ; Register r4-r11 are restored.

set_thread_and_privilege_modes:
    ; Also modify the control register to force use of thread mode as well
    ; For CM3 forward-compatibility, also set user mode.
    mrs r0, control
    mov r1, #0x03
    orr r0, r1
    control, r0

set_lr:
    ; Set up the link register such that on return, the code operates
    ; in thread mode using the PSP. To do this, we or 0x0D to the value stored
    ; in the lr by the exception hardware EXC_RETURN. Alternately, we could
    ; just force lr to be 0xFFFFFFFD (we know that's what we want from the
    ; hardware, anyway)
    mov  r0, #0x0D
    mov  r1, lr
    orr r0, r1

exit_exception:
    ; Return from the exception handler.
    ; The CPU will automagically unstack R0-R3, R12, PC, LR, and xPSR
    ; for us.  If all goes well, our thread will start execution at the
    ; entrypoint, with the us-specified argument.
    bx r0
```

On ARM Cortex parts, there's dedicated hardware that's used primarily to support RTOS (or RTOS-like) funcationlity. This functionality includes the SysTick timer, and the PendSV Exception. SysTick is used for a tick-based kernel timer, while the PendSV exception is used for performing context switches. In reality, it's a "special SVC" call that's designed to be lower-overhead, in that it isn't mux'd with a bunch of other system or application functionality.

So how do we go about actually implementing a context switch here? There are a lot of different parts involved, but it essentially comes down to 3 steps:

1) Saving the context.

```
Thread's top-of-stack value is stored, all registers are stacked.  We're
good to go!
```

2) Swap threads

```
We swap the Scheduler's "next" thread with the "current" thread.
```

3) Restore Context

```
This is more or less identical to what we did when restoring the first
context.  Some operations may be optimized for data already stored in
registers.
```

The code used to implement these steps on Cortex-M0 is presented below:

```
void PendSV_Handler(void)
{
    ASM(
    // Thread_SaveContext()
    " ldr r1, CURR_ \n"
    " ldr r1, [r1] \n "
    " mov r3, r1 \n "
    " add r3, #8 \n "

    //  Grab the psp and adjust it by 32 based on extra registers we're going
    // to be manually stacking.
    " mrs r2, psp \n "
```

```
        " sub r2, #32 \n "

        // While we're here, store the new top-of-stack value
        " str r2, [r3] \n "

        // And, while r2 is at the bottom of the stack frame, stack r7-r4
        " stmia r2!, {r4-r7} \n "

        // Stack r11-r8
        " mov r7, r11 \n "
        " mov r6, r10 \n "
        " mov r5, r9 \n "
        " mov r4, r8 \n "
        " stmia r2!, {r4-r7} \n "

        // Equivalent of Thread_Swap() - performs g_pstCurrent = g_pstNext
        " ldr r1, CURR_ \n"
        " ldr r0, NEXT_ \n"
        " ldr r0, [r0] \n"
        " str r0, [r1] \n"

        // Thread_RestoreContext()
        // Get the pointer to the next thread's stack
        " add r0, #8 \n "
        " ldr r2, [r0] \n "

        // Stack pointer is in r2, start loading registers from
        // the "manually-stacked" set
        // Start with r11-r8, since these can't be accessed directly.
        " add r2, #16 \n "
        " ldmia r2!, {r4-r7} \n "
        " mov r11, r7 \n "
        " mov r10, r6 \n "
        " mov r9, r5 \n "
        " mov r8, r4 \n "

        // After subbing R2 #16 manually, and #16 through ldmia, our PSP is where it
        // needs to be when we return from the exception handler
        " msr psp, r2 \n "

        // Pop manually-stacked R4-R7
        " sub r2, #32 \n "
        " ldmia r2!, {r4-r7} \n "

        // lr contains the proper EXC_RETURN value
        // we're done with the exception, so return back to newly-chosen thread
        " bx lr \n "
        " nop \n "

        // Must be 4-byte aligned.
        " NEXT_: .word g_pstNext \n"
        " CURR_: .word g_pstCurrent \n"
        );
}
```

Kernel Timers

ARM Cortex-M series microcontrollers each contain a SysTick timer, which was designed to facilitate a fixed-interval RTOS timer-tick. This timer is a precise 24-bit down-count timer, run at the main CPU clock frequency, that can be programmed to trigger an exception when the timer expires. The handler for this exception can thus be used to drive software timers throughout the system on a fixed interval.

Unfortunately, this hardware is extremely simple, and does not offer the flexibility of other timer hardware commonly implemented by MCU vendors - specifically a suitable timer prescalar that can be used to generate efficient, long-counting intervals. As a result, while the "generic" port of Mark3 for Cortex-M0 leverages the common SysTick timer interface, it only supports the tick-based version of the kernel's timer (note that specific Cortex-M0 ports such as the Atmel SAMD20 do have tickless timers).

Setting up a tick-based KernelTimer class to use the SysTick timer is, however, extremely easy, as is illustrated below:

```
void KernelTimer::Start(void)
{
    SysTick_Config(PORT_SYSTEM_FREQ / 1000); // 1KHz fixed clock...
    NVIC_EnableIRQ(SysTick_IRQn);
}

In this instance, the call to SysTick_Config() generates a 1kHz system-tick
signal, and the NVIC_EnableIRQ() call ensures that a SysTick exception is
generated for each tick.  All other functions in the Cortex version of the
KernelTimer class are essentially stubbed out (see the source for more details).
```

Note that the functions used in this call are part of the ARM Cortex
Microcontroller Software Interface Standard (cmsis), and are supplied by all
parts vendors selling Cortex hardware. This greatly simplifies the design
of our port-code, since we can be reasonably assured that these APIs will
work the same on all devices.

The handler code called when a SysTick exception occurs is basically the
same as on other platforms (such as AVR), except that we explicitly clear the
"exception pending" bit before returning. This is implemented in the
following code:

```cpp
\code{.cpp}
void SysTick_Handler(void)
{
#if KERNEL_USE_TIMERS
    TimerScheduler::Process();
#endif
#if KERNEL_USE_QUANTUM
    Quantum::UpdateTimer();
#endif

    // Clear the systick interrupt pending bit.
    SCB->ICSR |= SCB_ICSR_PENDSTCLR_Msk;
}
```

Critical Sections

A "critical section" is a block of code whose execution cannot be interrupted by means of context switches or an interrupt. In a traditional single-core operating system, it is typically implemented as a block of code where the interrupts are disabled - this is also the approach taken by Mark3. Given that every CPU has its own means of disabling/enabling interrupts, the implementation of the critical section APIs is also non-portable.

In the Cortex-M0 port, we implement the two critical section APIs (CS_ENTER() and CS_EXIT()) as function-like macros containing inline assembly. All uses of these calls are called in pairs within a function and must take place at the same level-of-scope. Also, as nesting may occur (critical section within a critical section), this must be taken into account in the code.

In general, CS_ENTER() performs the following tasks:

```
- Cache the current interrupt-enabled state within a local variable in the
thread's state
- Disable interrupts
.
```

Conversely, CS_EXIT() performs the following tasks:

```
- Read the original interrupt-enabled state from the cached value
- Restore interrupts to the original value
.
```

On Cortex-M series micrcontrollers, the PRIMASK special register contains a single status bit which can be used to enable/disable all maskable interrupts at once. This register can be read directly to examine or modify its state. For convenience, ARMv6m provides two instructions to enable/disable interrupts

- cpsid (disable interrupts) and cpsie (enable interrupts). Mark3 Implements these steps according to the following code:

```
//---------------------------------------------------------------------
#define CS_ENTER()    \
{   \
    K_ULONG __ulRegState;    \
    asm    ( \
    " mrs r0, PRIMASK \n"    \
    " mov %[STATUS], r0 \n" \
    " cpsid i \n "    \
    : [STATUS] "=r" (__ulRegState) \
    );

//---------------------------------------------------------------------
#define CS_EXIT() \
    asm    ( \
    " mov r0, %[STATUS] \n" \
    " msr primask, r0 \n"    \
```

```
    : \
    : [STATUS] "r" (__ulRegState) \
    ); \
}
```

Summary

In this section we have investigated how the main non-portable areas of the Mark3 RTOS are implemented on a Cortex-M0 microcontroller. Mark3 leverages all of the hardware blocks designed to enable RTOS functionality on ARM Cortex-M series microcontrollers: the SVC call provides the mechanism by which we start the kernel, the PendSV exception provides the necessary software interrupt, and the SysTick timer provides an RTOS tick. As a result, Mark3 is a perfect fit for these devices - and as a result of this approach, the same RTOS port code should work with little to no modification on all ARM Cortex-M parts.

We have discussed what functionality in the RTOS is not portable, and what interfaces must be implemented in order to complete a fully-functional port. The five specific areas which are non-portable (stack initialization, kernel startup/entry, kernel timers, context switching, and critical sections) have been discussed in detail, with the platform-specifc source provided as a practical reference to ARM-specific OS features, as well as Mark3's porting infrastructure. From this example (and the accompanying source), it should be possible for an experienced developers to create a port Mark3 to other microcontroller targets.

Chapter 11

Build System

In addition to providing a complete RTOS kernel with a variety of middleware, tests, and example code, Mark3 also provides a robust architecture to efficiently build these components.

The build system – including its design and use, are discussed in the following sections.

11.1   Introduction

As developers, we spend an awful lot of time talking about how our source code is written, but devote very little energy to what happens to the code after it's been written... aside from producing running executables. When I refer to "building better software", I'm not talking about writing code – I'm talking about the technologies and processes that can be applied to manipulate source into a variety of products, including libraries, applications, tests, documentation, and performance data.

For a lot of developers – embedded or otherwise – a typical build process might look something like this:

Open the IDE, load a project and click "build". Sometime later, check the output window and look to see that there aren't any red exclamation points to indicate build failure. Browse to your project's output folder to collect your prize: A brand new .elf file containing your new firmware! Click on the arrow to give it a quick run on your dev board, test it for a few minutes, and make sure it seems sane. Pass it off to the manufacturing guys to load it on the line, and move on. Next!

Okay, that's a bit of an exaggeration, but not too far-fetched; and not that much different from standard procedure at places I've worked in the past.

Indeed - I've come across many developers over the years who know about how their software gets built beyond the "black box" that turns their code from text to binaries with the click of the button – and they like it that way. It's entirely understandable, too. Developing from an IDE hides all those messy configuration details, command-line options, symbol definitions and environment variables that would otherwise take away from time spent actively churning out code. We all want to be more productive, of course, and it takes time to learn to make, or anything specific to an embedded toolchain.

And from a product delivery perspective, binaries are the ultimate work-products from a software team – these are the pieces that drive the microcontrollers, DSPs and CPUs in an embedded system. When its crunch time, try convincing management to back off on release date in order to ensure that documentation gets updated to reflect the as-built nature of a project. Or fix the gaps in test coverage. Or update wikis containing profiling and performance metrics. You get the picture.

But software is a living entity – it's constantly changing as it develops and is refined by individuals and teams. And source code is a medium that carries different information across multiple channels all at once – while one channel contains information about building an application, another contains information on building libraries. Another carriers information on testing, and another still provides documentation relevant to consumers of the code. While not as glamorous a role as the "living firmware", these pieces of critical metadata are absolutely necessary as they ensure that the firmware products maintain a degree of quality, performance, and conformance, and gives a degree of confidence before formal test and release activities take place.

This is especially necessary when developing for an organizations that is accountable for their development and documentation practices (for example, ISO shops), or to shareholders who expect the companies they support with their wallets to apply engineering rigour to their products.

But getting the kind of flexibility required to produce these alternative work products form the "example IDE" is not trivial, and can be difficult to apply consistently from project-to-project/IDE-to-IDE. Automating these test and documentation tasks should be considered mandatory if you care about making the most of your development hours; manually generating and updating documentation, tests, and profiling results wastes time that you could be spending solving the right kinds of problems.

The good news, though, is that using common tools available on any modern OS, you can create frameworks that make these tasks for any project, on any toolchain providing command-line tools. With a bit of make, shell-script, and python, you can automate any number of build processes in a way that yields consistent, reliable results that are transferrable from project to project.

This is the approach taken in the Mark3 project, which integrates build, testing, profiling, documentation and release processes together in order to produce predictable, verifiable, output that can be validated against quality gates prior to formal testing and release. Only code revisions that pass all quality gate can be released. In the following sections, we'll explore the phased build approach, and how it's used by the Mark3 project.

11.2 Mark3 Build Process Overview

Building software is by and large a serial process, as outputs from each build step are required in subsequent steps. We start from our source code, scripts, and makefiles, configure our environment, and use our tools to turn the source code from one form to another, leveraging the outputs from each stage in the generation of further work products – whether it be creating binaries, running tests, or packaging artifacts for release.

To simplify the design and illustrate the concepts involved, we can break down these serial process into the following distinct phases:

- Pre-build – Environment configuration, target selection, and header-file staging

- Build – Compiling libraries, and building binaries for applications and tests

- Test + Profiling - Running unit tests, integration tests, profiling code

- Release – Generation of documentation from source code and test results, packaging of build artifacts and headers

Each phase and associated activities are described in detail in the following subsections.

11.2.1 Pre-Build Phase:

Target Selection

Inputs: CPU Architecture, Variant, Toolchain variables Outputs: Environment, makefile configuration

In this phase, we select the runtime environment and configure all environment-specific variables. Specifying environment variables at this phase ensures that when the build scripts are run, the correct makefiles, libraries, binaries, and config files are used when generating outputs. This can also be used to ensure that common build setting are applied to all platform specific binaries, including optimization levels, debug symbols, linker files, and CPU flags.

Staging Headers

Inputs: All files with a .h extension, located in library or binary project /public folders Output: Headers copied to a common staging directory

In this step, header files from all platform libraries are copied to a common staging directory referenced by the build system.

This simplifies makefiles and build scripts, ensuring only a single include directory needs to be specified to gain access to all common platform libraries. This keeps library and application code clean, as relative paths can be

completely avoided. As an added benefit, these headers can later be deployed with the corresponding libraries to customers, giving them access to a set of pre-compiled libraries with APIs, but without providing the source.

11.2.2 Build Phase

Building Libraries

Input: Source code for all common libraries, staged headers Output: Static libraries that can be linked against applications Gate: All mandatory libraries must be built successfully

The project root directory is scanned recursively for directories containing makefiles. When a makefile is found in the root of a subdirectory and a library tag is encountered (in Mark3, this corresponds to the declaration "IS_LIB=1"), the project is built using the library-specific make commands for the platform. Libraries can reference other libraries implicitly, and include headers from the common include directory. Since references are resolved when building executable binary images, the executable projects are responsible for including the dependent libs.

Building Binaries

Input: Source code for individual applications, precompiled libraries, staged headers Output: Executable application and test binaries Gate: All mandatory binaries (applications and tests) must be built successfully

The project root directory is scanned recursively for directories containing makefiles. When a makefile is found in the root of a subdirectory and a binary tag is encountered (in Mark3, this corresponds to the declaration "IS_AP↩P=1"), the project is built using the executable-specific make commands for the platform. Applications can reference all platform and toolchain libraries, and include headers from the common include directory. Care must be taken to ensure that all library depenencies are explicitly specified in the application's makefile's list.

This step will fail if necessary dependencies are not met (i.e. required libraries failed to build in a prior step).

Static Analysis:

Input: Source code for libraries/binaries Output: Static source analysis output Gate: N/A

Static analysis tools such as clang, klocwork, and lint can be run on the source to ensure that there are no critical or catastrophic problems (null pointer exceptions, variables used before initialization, incorrect argument usage, etc.) that wouldn't necessarily be caught at compile-time. Since tool availability and configurability varies, this isn't something that is enforced in the Mark3 builds. A user may opt to use clang to perform static code analysis on the build, however. The part-specific makefile contains a CLANG environment variable for this purpose.

Potential quality gates could be set up such that a failure during static analysis aborts the rest of the build.

Test + Profiling Sanity Tests

Input: Executable test binaries, CPU simulator/embedded target system Output: Text output indicating test pass/failure status

11.2.3 Test and Profile

Unit Tests

Input: Executable test binaries, CPU simulator/embedded target system Output: Text output indicating test pass/failure status

Code Performance Profiling

Input: Executable test binaries, CPU simulator/embedded target system Output: Text output containing critical code performance metrics

Code Size Profiling

Input: Precompiled static libraries and binaries Output: Text output containing critical code size metrics

11.2.4   Release

Documentation

Input: Library source code and headers, commented with Doxygen tags, Profiling results, Test results Output: Doxygen-generated HTML and PDF documentation

Packaging

Input: Static libraries and application/test binaries, staged headers, compiled documentation Output: Archive (.zip) containing relevant build outputs

Chapter 12

Mark3C - C-language API bindings for the Mark3 Kernel.

Mark3 now includes an optional additional library with C language bindings for all core kernel APIs, known as Mark3C.

This library alllows applications to be written in C, while still enjoying all of the benefits of the clean, modular design of the core RTOS kernel.

The C-language Mark3C APIs map directly to their Mark3 counterparts using a simple set of conventions, documented below. As a result, explicit API documentation for Mark3C is not necessary, as the functions map 1-1 to their C++ counterparts.

12.1 API Conventions

1) Static Methods:

```
<ClassName>::<MethodName>()   Becomes      <ClassName>_<MethodName>()
i.e. Kernel::Start()          Becomes      Kernel_Start()
```

2) Kernel Object Methods:

In short, any class instance is represented using an object handle, and is always passed into the relevant APIs as the first argument. Further, any method that returns a pointer to an object in the C++ implementation now returns a handle to that object.

```
<Object>.<MethodName>(<args>) Becomes      <ClassName>_<MethoodName>(<ObjectHandle>, <args>)

i.e. clAppThread.Start()      Becomes      Thread_Start(hAppThread)
```

3) Overloaded Methods:

a) Methods overloaded with a Timeout parameter:

```
<Object>.<MethodName>(<args>) Becomes      <ClassName>_Timed<MethodName>(<ObjectHandle>, <args>)

i.e. clSemaphore.Wait(1000)   Becomes      Semaphore_Wait(hSemaphore, 1000)
```

b) Methods overloaded based on number of arguments:

```
<Object>.<MethodName>()               Becomes      <ClassName>_<MethodName>(<ObjectHandle>)
<Object>.<MethodName>(<arg1>)         Becomes      <ClassName>_<MethodName>1(<ObjectHandle>, <arg1>)
<Object>.<MethodName>(<arg1>, <arg2>) Becomes      <ClassName>_<MethodName>2(<ObjectHandle>, <arg1>, <arg2>
```

```
<ClassName>::<MethodName>()                 Becomes      <ClassName>_<MethodName>(<ObjectHandle>)
<ClassName>::<MethodName>(<arg1>)           Becomes      <ClassName>_<MethodName>1(<ObjectHandle>, <arg1>)
<ClassName>::<MethodName>(<arg1>, <arg2>)   Becomes      <ClassName>_<MethodName>2(<ObjectHandle>, <arg1>, <arg2>
```

c) Methods overloaded base on parameter types:

```
<Object>.<MethodName>(<arg type_a>)         Becomes      <ClassName>_<MethodName><type_a>(<ObjectHandle>, <arg ty
<Object>.<MethodName>(<arg type_b>)         Becomes      <ClassName>_<MethodName><type_b>(<ObjectHandle>, <arg ty
<ClassName>::<MethodName>(<arg type_a>)     Becomes      <ClassName>_<MethodName><type_a>(<arg type a>)
<ClassName>::<MethodName>(<arg type_b>)     Becomes      <ClassName>_<MethodName><type_b>(<arg type b>)
```

d) Allocate-once memory allocation APIs

```
AutoAlloc::New<ObjectName>                  Becomes       Alloc_<ObjectName>
AutoAlloc::Allocate(uint16_t u16Size_)      Becomes       AutoAlloc(uint16_t u16Size_)
```

12.2 Allocating Objects

Aside from the API name translations, the object allocation scheme is the major different between Mark3C and Mark3. Instead of instantiating objects of the various kernel types, kernel objects must be declared using Declaration macros, which serve the purpose of reserving memory for the kernel object, and provide an opaque handle to that object memory. This is the case for statically-allocated objects, and objects allocated on the stack.

Example: Declaring a thread

```
#include "mark3c.h"

// Statically-allocated
DECLARE_THREAD(hMyThread1);
...

// On stack
int main(void)
{
    DECLARE_THREAD(hMyThread2);
    ...
}
```

Where:

```
hMyThread1 - is a handle to a statically-allocated thread
hMyThread2 - is a handle to a thread allocated from the main stack.
```

Alternatively, the AutoAlloc APIs can be used to dynamically allocate objects, as demonstrated in the following example.

```
void Allocate_Example(void)
{
    Thread_t hMyThread = AutoAlloc_Thread();

    Thread_Init(hMyThread, awMyStack, sizeof(awMyStack), 1, MyFunction, 0);
}
```

Note that the relevant kernel-object Init() function *must* be called prior to using any kernel object, whether or not they have been allocated statically, or dynamically.

12.3 Drivers in Mark3C

Because the Mark3 driver framework makes extensive use of inheritence and virtual functions in C++, it is difficult to wrap for use with C. In addition, all derived drivers types would still need to have their custom interfaces wrapped

with C-language bindings in order to be accessible from C, which is cumbersome and inelegant, and duplicates large portions of code. As a result, it's probably less work to write a Mark3C specific driver module with a similar interface to Mark3, on which drivers can be ported where necessary, or implemented directly on for efficiency. The APIs presented in driver3c.h provide such an interface for use in Mark3c.

Chapter 13

Release Notes

13.1 R4 Release

- New: C-language bindings for Mark3 kernel (mark3c library)

- New: Support for ARM Cortex-M3 and Cortex-M4 (floating point) targets

- New: Support for Atmel AVR atmega2560 and arduino pro mega

- New: Full-featured, lightweight heap implementation

- New: Mailbox IPC class

- New: Notification object class

- New: lighweight tracelogger/instrumentation implementation (buffalogger), with sample parser

- New: High-performance AVR Software UART implementation

- New: Allocate-once "AutoAlloc" memory allocator

- New: Fixed-time blocking/unblocking operations added to ThreadList/Blocking class

- Placement-new supported for all kernel objects

- Scheduler now supports up to 1024 levels of thread priority, up from 8 (configurable at build-time)

- Kernel now uses stdint.h types for standard integers (instead of K_CHAR, K_ULONG, etc.)

- Greatly expanded documentation, with many new examples covering all key kernel features

- Expanded unit test coverage on AVR

- Updated build system and scripts for easier kernel configuration

- Updated builds to only attempt to build tests for supported platforms

13.2 R3 Release

- New: Added support for MSP430 microcontrollers

- New: Added Kernel Idle-Function hook to eliminate the need for a dedicated idle-thread (where supported)

- New: Support for kernel-aware simulation and testing via flAVR AVR simulator

- Updated AVR driver selection

- General bugfixes and maintenance

- Expanded documentation and test coverage

13.3 R2

- Experimental release, using a "kernel transaction queue" for serializing kernel calls

- Works as a proof-of-concept, but abandoned due to overhead of the transaction mechanism in the general case.

13.4 R1 - 2nd Release

- New: Added support for ARM Cortex-M0 targets

- New: Added support for variuos AVR targets

- New: Timers now support a "tolerance" parameter for grouping timers with close expiry times

- Expanded scripts and auotmation used in build/test

- Updated and expanded graphics APIs

- Large number of bugfixes

13.5 R1 - 1st Release

- Initial release, with support for AVR microcontrollers

Chapter 14

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.

14.1 Date Performed

Wed Sep 21 23:08:12 EDT 2016

14.2 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)
```

14.3 Profiling Results

- Semaphore Initialization: 39 cycles (averaged over 42 iterations)
- Semaphore Post (uncontested): 111 cycles (averaged over 42 iterations)
- Semaphore Pend (uncontested): 69 cycles (averaged over 42 iterations)
- Semaphore Flyback Time (Contested Pend): 1639 cycles (averaged over 42 iterations)
- Mutex Init: 215 cycles (averaged over 43 iterations)
- Mutex Claim: 177 cycles (averaged over 43 iterations)
- Mutex Release: 135 cycles (averaged over 42 iterations)
- Thread Initialize: 8273 cycles (averaged over 42 iterations)
- Thread Start: 791 cycles (averaged over 42 iterations)
- Context Switch: 135 cycles (averaged over 42 iterations)
- Thread Schedule: 56 cycles (averaged over 42 iterations)

Chapter 15

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.

15.1   Information

Subversion Repository Information:

- Repository Root: svn+ssh://m0slevin.code.sf.net/p/mark3/source

- Revision: 362

- URL: svn+ssh://m0slevin.code.sf.net/p/mark3/source/trunk/embedded Relative URL: $^\wedge$/trunk/embedded

Date Profiled: Wed Sep 21 23:08:13 EDT 2016

15.2   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.

15.3   Profiling Results

Mark3 Module Size Report:

```
- Allocate-once Heap.............................. : 0 Bytes
- Synchronization Objects - Base Class............ : 126 Bytes
- Device Driver Framework (including /dev/null)... : 212 Bytes
- Synchronization Object - Event Flag............. : 724 Bytes
- Fundamental Kernel Linked-List Classes.......... : 450 Bytes
- Message-based IPC............................... : 384 Bytes
- Mutex (Synchronization Object).................. : 716 Bytes
- Notification Blocking Object.................... : 524 Bytes
- Performance-profiling timers.................... : 480 Bytes
- 2D Priority Map Object - Scheduler.............. : 116 Bytes
- Round-Robin Scheduling Support.................. : 272 Bytes
```

– Thread Scheduling............................... : 318 Bytes
– Semaphore (Synchronization Object).............. : 526 Bytes
– Mailbox IPC Support............................. : 856 Bytes
– Thread Implementation........................... : 1641 Bytes
– Fundamental Kernel Thread-list Data Structures.. : 250 Bytes
– Mark3 Kernel Base Class......................... : 145 Bytes
– Software Timer Kernel Object.................... : 452 Bytes
– Software Timer Management....................... : 621 Bytes
– Runtime Kernel Trace Implementation............. : 0 Bytes
– Atmel AVR – Kernel Aware Simulation Support...... : 190 Bytes
– Atmel AVR – Basic Threading Support.............. : 614 Bytes
– Atmel AVR – Kernel Interrupt Implemenation....... : 56 Bytes
– Atmel AVR – Kernel Timer Implementation.......... : 302 Bytes
– Atmel AVR – Profiling Timer Implementation....... : 216 Bytes

Mark3 Kernel Size Summary:

– Kernel   : 3046 Bytes
– Synchronization Objects : 2350 Bytes
– Port   : 2758 Bytes
– Features   : 2037 Bytes
– Total Size   : 10191 Bytes

Chapter 16

Hierarchical Index

16.1 Class Hierarchy

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

Chapter 17

Class Index

17.1 Class List

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

Chapter 18

File Index

18.1 File List

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

Chapter 19

Class Documentation

19.1 BlockingObject Class Reference

Class implementing thread-blocking primatives.

```
#include <blocking.h>
```

Inheritance diagram for BlockingObject:

```
                    ┌─────────────────┐
                    │  BlockingObject  │
                    └─────────────────┘
          ┌────────────┬─────┴──────┬─────────────┐
   ┌──────────┐  ┌──────────┐  ┌──────────┐  ┌──────────┐
   │ EventFlag │  │  Mutex   │  │  Notify  │  │ Semaphore │
   └──────────┘  └──────────┘  └──────────┘  └──────────┘
```
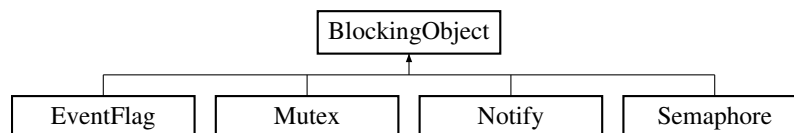
Protected Member Functions

- void Block (Thread ∗pclThread_)

    Block.

- void BlockPriority (Thread ∗pclThread_)

    BlockPriority.

- void UnBlock (Thread ∗pclThread_)

    UnBlock.

Protected Attributes

- ThreadList m_clBlockList

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

19.1.1 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.

19.1.2 Member Function Documentation

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

Block.

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.

Parameters

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

Definition at line 41 of file blocking.cpp.

19.1.2.2 void BlockingObject::BlockPriority (Thread ∗ *pclThread_*) `[protected]`

BlockPriority.

Same as Block(), but ensures that threads are added to the block-list in priority-order, which optimizes the unblock procedure.

Parameters

*pclThread_*	Pointer to the Thread to Block.

Definition at line 57 of file blocking.cpp.

19.1.2.3 void BlockingObject::UnBlock (Thread ∗ *pclThread_*) `[protected]`

UnBlock.

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

Parameters

*pclThread_*	Pointer to the thread to unblock.

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

Definition at line 73 of file blocking.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/blocking.h
- /home/moslevin/mark3-source/embedded/kernel/blocking.cpp

19.2 CircularLinkList Class Reference

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

```
#include <ll.h>
```

Inheritance diagram for CircularLinkList:

LinkList

CircularLinkList

ThreadList
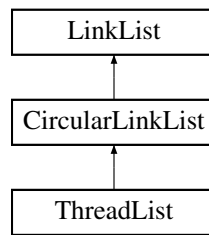
Public Member Functions

- void Add (LinkListNode ∗node_)

    Add the linked list node to this linked list.

- void Remove (LinkListNode ∗node_)

    Remove.

- void PivotForward ()

    PivotForward.

- void PivotBackward ()

    PivotBackward.

- void InsertNodeBefore (LinkListNode ∗node_, LinkListNode ∗insert_)

    InsertNodeBefore.

Additional Inherited Members

19.2.1   Detailed Description

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

Definition at line 187 of file ll.h.

19.2.2   Member Function Documentation

19.2.2.1   void CircularLinkList::Add (LinkListNode ∗ node_)

Add the linked list node to this linked list.

Parameters

*node_*	Pointer to the node to add

Definition at line 97 of file ll.cpp.

19.2.2.2   void CircularLinkList::InsertNodeBefore (LinkListNode ∗ node_, LinkListNode ∗ insert_)

InsertNodeBefore.

Insert a linked-list node into the list before the specified insertion point.

Parameters

*node_*	Node to insert into the list
*insert_*	Insert point.

Definition at line 171 of file ll.cpp.

19.2.2.3   void CircularLinkList::PivotBackward (   )

PivotBackward.

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

Definition at line 162 of file ll.cpp.

19.2.2.4   void CircularLinkList::PivotForward (   )

PivotForward.

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

Definition at line 153 of file ll.cpp.

19.2.2.5   void CircularLinkList::Remove (LinkListNode ∗ node_)

Remove.

Add the linked list node to this linked list

Parameters

*node_*	Pointer to the node to remove

Definition at line 119 of file ll.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/ll.h
- /home/moslevin/mark3-source/embedded/kernel/ll.cpp

19.3   DevNull Class Reference

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

Inheritance diagram for DevNull:

```
        LinkListNode
             ▲
             │
          Driver
             ▲
             │
         DevNull
```
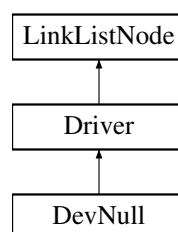
Public Member Functions

- virtual void Init ()

    Init.
- virtual uint8_t Open ()

    Open.
- virtual uint8_t Close ()

    Close.
- virtual uint16_t Read (uint16_t u16Bytes_, uint8_t ∗pu8Data_)

    Read.

- virtual uint16_t [Write](uint16_t u16Bytes_, uint8_t *pu8Data_)

    Write.

- virtual uint16_t [Control](uint16_t u16Event_, void *pvDataIn_, uint16_t u16SizeIn_, void *pvDataOut_↩
, uint16_t u16SizeOut_)

    Control.

Additional Inherited Members

19.3.1   Detailed Description

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

Definition at line 46 of file driver.cpp.

19.3.2   Member Function Documentation

19.3.2.1   virtual uint8_t DevNull::Close ( ) `[inline],[virtual]`

Close.

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.

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

Control.

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 55 of file driver.cpp.

19.3.2.3 virtual void DevNull::Init ( ) `[inline],[virtual]`

Init.

Initialize a driver, must be called prior to use

Implements [Driver].

Definition at line 49 of file [driver.cpp].

19.3.2.4 virtual uint8_t DevNull::Open ( ) `[inline],[virtual]`

Open.

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].

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

Read.

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 52 of file [driver.cpp].

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

Write.

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 53 of file driver.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/driver.cpp

19.4 DoubleLinkList Class Reference

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

```
#include <ll.h>
```

Inheritance diagram for DoubleLinkList:

```
┌─────────────────┐
│     LinkList     │
└─────────────────┘
         ▲
┌─────────────────┐
│  DoubleLinkList  │
└─────────────────┘
         ▲
┌─────────────────┐
│     TimerList    │
└─────────────────┘
```
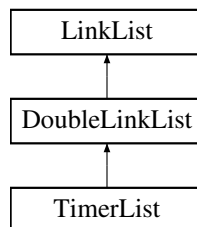
Public Member Functions

- DoubleLinkList ()

    DoubleLinkList.
- void Add (LinkListNode ∗node_)

    Add.
- void Remove (LinkListNode ∗node_)

    Remove.

Additional Inherited Members

19.4.1 Detailed Description

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

Definition at line 149 of file ll.h.

19.4.2 Constructor & Destructor Documentation

19.4.2.1 DoubleLinkList::DoubleLinkList ( ) [inline]

DoubleLinkList.

Default constructor - initializes the head/tail nodes to NULL

Definition at line 158 of file ll.h.

19.4.3   Member Function Documentation

19.4.3.1   void DoubleLinkList::Add (LinkListNode ∗ *node_*)

Add.

Add the linked list node to this linked list

Parameters

*node_*	Pointer to the node to add

Definition at line 47 of file ll.cpp.

19.4.3.2   void DoubleLinkList::Remove (LinkListNode ∗ *node_*)

Remove.

Add the linked list node to this linked list

Parameters

*node_*	Pointer to the node to remove

Definition at line 68 of file ll.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/ll.h
- /home/moslevin/mark3-source/embedded/kernel/ll.cpp

19.5   Driver Class Reference

Base device-driver class used in hardware abstraction.

```
#include <driver.h>
```

Inheritance diagram for Driver:
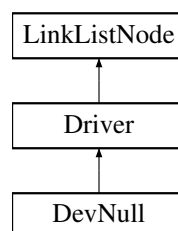
Public Member Functions

- virtual void Init ()=0

    Init.
- virtual uint8_t Open ()=0

    Open.
- virtual uint8_t Close ()=0

    Close.
- virtual uint16_t Read (uint16_t u16Bytes_, uint8_t ∗pu8Data_)=0

    Read.
- virtual uint16_t Write (uint16_t u16Bytes_, uint8_t ∗pu8Data_)=0

Write.

- virtual uint16_t Control (uint16_t u16Event_, void ∗pvDataIn_, uint16_t u16SizeIn_, void ∗pvDataOut_↩
, uint16_t u16SizeOut_)=0

    Control.

- void SetName (const char ∗pcName_)

    SetName.

- const char ∗ GetPath ()

    GetPath.

Private Attributes

- const char ∗ m_pcPath

    string pointer that holds the driver path (name)

Additional Inherited Members

19.5.1 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.

19.5.2 Member Function Documentation

19.5.2.1 virtual uint8_t Driver::Close ( ) `[pure virtual]`

Close.

Close a previously-opened device driver.

Returns

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

Implemented in DevNull.

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

Control.

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].

19.5.2.3   const char∗ Driver::GetPath (   ) `[inline]`

GetPath.

Returns a string containing the device path.

Returns

    pcName_ Return the string constant representing the device path

Definition at line 221 of file [driver.h].

19.5.2.4   virtual void Driver::Init (   ) `[pure virtual]`

Init.

Initialize a driver, must be called prior to use

Implemented in [DevNull].

19.5.2.5   virtual uint8_t Driver::Open (   ) `[pure virtual]`

Open.

Open a device driver prior to use.

Returns

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

Implemented in [DevNull].

19.5.2.6   virtual uint16_t Driver::Read (uint16_t *u16Bytes_,* uint8_t ∗ *pu8Data_*) `[pure virtual]`

Read.

Read a specified number of bytes from the device into a specific buffer. Depending on the driver-specific imple-
mentation, 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].

19.5.2.7 void Driver::SetName (const char ∗ _pcName__) `[inline]`

SetName.

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 213 of file driver.h.

19.5.2.8 virtual uint16_t Driver::Write (uint16_t _u16Bytes__, uint8_t ∗ _pu8Data__) `[pure virtual]`

Write.

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/moslevin/mark3-source/embedded/kernel/public/driver.h

19.6 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 ()
    Init.
- static void Add (Driver ∗pclDriver_)
    Add.
- static void Remove (Driver ∗pclDriver_)
    Remove.
- static Driver ∗ FindByPath (const char ∗m_pcPath)
    FindByPath.

Static Private Attributes

- static DoubleLinkList m_clDriverList
    LinkedList object used to implementing the driver object management.

19.6.1 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 232 of file driver.h.

19.6.2 Member Function Documentation

19.6.2.1 static void DriverList::Add (Driver ∗ *pclDriver_*) `[inline],[static]`

Add.

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

Parameters

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

Examples:

> buffalogger/main.cpp.

Definition at line 252 of file driver.h.

19.6.2.2 Driver ∗ DriverList::FindByPath (const char ∗ *m_pcPath*) `[static]`

FindByPath.

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 104 of file driver.cpp.

19.6.2.3 void DriverList::Init ( ) `[static]`

Init.

Initialize the list of drivers. Must be called prior to using the device driver library.

Definition at line 95 of file driver.cpp.

19.6.2.4 static void DriverList::Remove (Driver ∗ *pclDriver_*) `[inline],[static]`

Remove.

Remove a driver from the global driver list.

Parameters

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

Definition at line 261 of file driver.h.

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

- /home/moslevin/mark3-source/embedded/kernel/public/driver.h
- /home/moslevin/mark3-source/embedded/kernel/driver.cpp

19.7 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:

```
┌─────────────────┐
│  BlockingObject  │
└─────────────────┘
         ▲
         │
┌─────────────────┐
│    EventFlag     │
└─────────────────┘
```
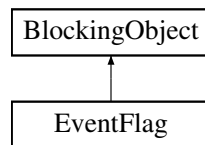
Public Member Functions

- 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

19.7.1 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.

Examples:

      lab7_events/main.cpp.

Definition at line 46 of file eventflag.h.

19.7.2 Member Function Documentation

19.7.2.1 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

Examples:

      lab7_events/main.cpp.

Definition at line 292 of file eventflag.cpp.

19.7.2.2 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 301 of file eventflag.cpp.

19.7.2.3 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.

Examples:

      lab7_events/main.cpp.

Definition at line 186 of file eventflag.cpp.

19.7.2.4 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

Examples:

      lab7_events/main.cpp.

Definition at line 168 of file eventflag.cpp.

19.7.2.5 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 179 of file eventflag.cpp.

19.7.2.6 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 84 of file eventflag.cpp.

19.7.2.7 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 76 of file eventflag.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/eventflag.h
- /home/moslevin/mark3-source/embedded/kernel/eventflag.cpp

19.8 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.
- PORT_PRIO_TYPE m_uXPriority

    Default priority of the thread.
- PORT_PRIO_TYPE m_uXCurPriority

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

    Enum indicating the thread's current state.

19.8.1 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 483 of file thread.h.

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

- /home/moslevin/mark3-source/embedded/kernel/public/thread.h

19.9 GlobalMessagePool Class Reference

Implements a list of message objects shared between all threads.

`#include <message.h>`

Static Public Member Functions

- static void Init ()

    Init.
- static void Push (Message *pclMessage_)

    Push.
- static Message * Pop ()

    Pop.
- static Message * GetHead ()

    GetHead.
- static MessagePool * GetPool ()

    GetPool.

Static Private Attributes

- static Message m_aclMessagePool [GLOBAL_MESSAGE_POOL_SIZE]

    Array of message objects that make up the message pool.

19.9.1 Detailed Description

Implements a list of message objects shared between all threads.

Definition at line 208 of file message.h.

19.9.2 Member Function Documentation

19.9.2.1 Message * GlobalMessagePool::GetHead ( ) `[static]`

GetHead.

Return a pointer to the first element in the message list

Returns

    Pointer to head message element, or NULL if empty

Definition at line 112 of file message.cpp.

19.9.2.2 MessagePool * GlobalMessagePool::GetPool ( ) `[static]`

GetPool.

Get the pointer to the underlying message pool object

Returns

    Pointer to message pool.

Definition at line 118 of file message.cpp.

19.9.2.3 void GlobalMessagePool::Init (void) `[static]`

Init.

Initialize the message queue prior to use

Definition at line 89 of file message.cpp.

19.9.2.4 Message ∗ GlobalMessagePool::Pop ( ) `[static]`

Pop.

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

Examples:

> lab8_messages/main.cpp.

Definition at line 106 of file message.cpp.

19.9.2.5 void GlobalMessagePool::Push (Message ∗ *pclMessage_*) `[static]`

Push.

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

Examples:

> lab8_messages/main.cpp.

Definition at line 100 of file message.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/message.h
- /home/moslevin/mark3-source/embedded/kernel/message.cpp

19.10 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 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.

- static bool IsStarted ()

    IsStarted.

- static void SetPanic (PanicFunc_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 (IdleFunc_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 void SetThreadCreateCallout (ThreadCreateCallout_t pfCreate_)

    SetThreadCreateCallout.

- static void SetThreadExitCallout (ThreadExitCallout_t pfExit_)

    SetThreadExitCallout.

- static void SetThreadContextSwitchCallout (ThreadContextCallout_t pfContext_)

    SetThreadContextSwitchCallout.

- static ThreadCreateCallout_t GetThreadCreateCallout (void)

    GetThreadCreateCallout.

- static ThreadExitCallout_t GetThreadExitCallout (void)

    GetThreadExitCallout.

- static ThreadContextCallout_t GetThreadContextSwitchCallout (void)

    GetThreadContextSwitchCallout.

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 PanicFunc_t m_pfPanic

    set panic function

- static IdleFunc_t m_pfIdle

    set idle function

- static FakeThread_t m_clIdle

    Idle thread object (note: not a real thread)

- static ThreadCreateCallout_t m_pfThreadCreateCallout

    Function to call on thread creation.

- static ThreadExitCallout_t m_pfThreadExitCallout

    Function to call on thread exit.

- static ThreadContextCallout_t m_pfThreadContextCallout

    Function to call on context switch.

19.10.1 Detailed Description

Class that encapsulates all of the kernel startup functions.

Definition at line 44 of file kernel.h.

19.10.2 Member Function Documentation

19.10.2.1 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 122 of file kernel.h.

19.10.2.2 static ThreadContextCallout_t Kernel::GetThreadContextSwitchCallout (void) `[inline],[static]`

GetThreadContextSwitchCallout.

Return the current function called on every Thread::ContextSwitchSWI()

Returns

Pointer to the currently-installed callout function, or NULL if not set.

Definition at line 190 of file kernel.h.

19.10.2.3 static ThreadCreateCallout_t Kernel::GetThreadCreateCallout (void) `[inline],[static]`

GetThreadCreateCallout.

Return the current function called on every Thread::Init();

Returns

Pointer to the currently-installed callout function, or NULL if not set.

Definition at line 172 of file kernel.h.

19.10.2.4 static ThreadExitCallout_t Kernel::GetThreadExitCallout (void) `[inline],[static]`

GetThreadExitCallout.

Return the current function called on every Thread::Exit();

Returns

Pointer to the currently-installed callout function, or NULL if not set.

Definition at line 181 of file kernel.h.

19.10.2.5   void 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.

Examples:

> buffalogger/main.cpp, lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_idle_function/main.cpp, lab3_round_robin/main.cpp, lab4_semaphores/main.cpp, lab5_mutexes/main.↩ cpp, lab6_timers/main.cpp, lab7_events/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_↩ threads/main.cpp.

Definition at line 67 of file kernel.cpp.

19.10.2.6   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 90 of file kernel.h.

19.10.2.7   static bool Kernel::IsStarted ( ) `[inline],[static]`

IsStarted.

Returns

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

Definition at line 77 of file kernel.h.

19.10.2.8   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 110 of file kernel.cpp.

19.10.2.9   static void Kernel::SetIdleFunc (IdleFunc_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

Examples:

> lab2_idle_function/main.cpp.

Definition at line 103 of file kernel.h.

19.10.2.10   static void Kernel::SetPanic (PanicFunc_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.

19.10.2.10   static void Kernel::SetPanic (PanicFunc_t *pfPanic_* **)**   `[inline],[static]`

Parameters

*pfPanic_*	Panic function pointer

Definition at line 85 of file kernel.h.

19.10.2.11   static void Kernel::SetThreadContextSwitchCallout (ThreadContextCallout_t *pfContext_* **)** `[inline],` `[static]`

SetThreadContextSwitchCallout.

Set a function to be called on each context switch.

A callout is only executed if this method has been called to set a valid handler function.

Parameters

*pfContext_*	Pointer to a function to call on context switch

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 159 of file kernel.h.

19.10.2.12   static void Kernel::SetThreadCreateCallout (ThreadCreateCallout_t *pfCreate_* **)** `[inline],[static]`

SetThreadCreateCallout.

Set a function to be called on creation of a new thread. This callout is executed on the successful completion of a Thread::Init() call. A callout is only executed if this method has been called to set a valid handler function.

Parameters

*pfCreate_*	Pointer to a function to call on thread creation

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 136 of file kernel.h.

19.10.2.13   static void Kernel::SetThreadExitCallout (ThreadExitCallout_t *pfExit_* **)** `[inline],[static]`

SetThreadExitCallout.

Set a function to be called on thread exit. This callout is executed from the beginning of Thread::Exit().

A callout is only executed if this method has been called to set a valid handler function.

Parameters

*pfCreate_*	Pointer to a function to call on thread exit

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 148 of file kernel.h.

19.10.2.14 **void Kernel::Start (void)** `[static]`

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. The exception to this is if the system is configured to use the threadless idle hook, in which case the kernel is allowed to run without any ready threads.

Examples:

buffalogger/main.cpp, lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_idle_function/main.cpp, lab3_round_robin/main.cpp, lab4_semaphores/main.cpp, lab5_mutexes/main.↩ cpp, lab6_timers/main.cpp, lab7_events/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_↩ threads/main.cpp.

Definition at line 101 of file kernel.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/kernel.h
- /home/moslevin/mark3-source/embedded/kernel/kernel.cpp

19.11 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_, Kernel↩
AwareCommand_t eCmd_)

    Trace_i.

19.11.1 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 64 of file kernelaware.h.

19.11.2 Member Function Documentation

19.11.2.1 void KernelAware::ExitSimulator (void) `[static]`

ExitSimulator.

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

Definition at line 109 of file kernelaware.cpp.

19.11.2.2 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 154 of file kernelaware.cpp.

19.11.2.3 void KernelAware::Print (const char ∗ *szStr_*) `[static]`

Print.

Instruct the kernel-aware simulator to print a char string

Parameters

*szStr_*	

Examples:

lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_idle_function/main.↩
cpp, lab3_round_robin/main.cpp, lab4_semaphores/main.cpp, lab5_mutexes/main.cpp, lab6_timers/main.cpp,
lab7_events/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_threads/main.cpp.

Definition at line 145 of file kernelaware.cpp.

19.11.2.4   void KernelAware::ProfileInit (const char ∗ *szStr_*) `[static]`

ProfileInit.

Initializes the kernel-aware profiler. This function instructs the kernel-aware simulator to reset its accounting vari-
ables, 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 82 of file kernelaware.cpp.

19.11.2.5   void KernelAware::ProfileReport (void) `[static]`

ProfileReport.

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

Definition at line 103 of file kernelaware.cpp.

19.11.2.6   void KernelAware::ProfileStart (void) `[static]`

ProfileStart.

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

Definition at line 91 of file kernelaware.cpp.

19.11.2.7   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 97 of file kernelaware.cpp.

19.11.2.8   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

Examples:

lab11_mailboxes/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_threads/main.cpp.

Definition at line 115 of file kernelaware.cpp.

19.11.2.9   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 121 of file kernelaware.cpp.

19.11.2.10   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 126 of file kernelaware.cpp.

19.11.2.11   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 132 of file kernelaware.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/kernelaware.h
- /home/moslevin/mark3-source/embedded/kernel/kernelaware.cpp

19.12   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 simluator.*

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.

19.12.1 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/moslevin/mark3-source/embedded/kernel/kernelaware.cpp

19.13 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)

    Config.
- static void Start (void)

    Start.
- static void Stop (void)

    Stop.
- static void Clear (void)

    Clear.
- static void Trigger (void)

    Trigger.
- static uint8_t DI ()

    DI.
- static void RI (bool bEnable_)

    RI.

19.13.1 Detailed Description

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

Definition at line 31 of file kernelswi.h.

19.13.2 Member Function Documentation

19.13.2.1 void KernelSWI::Clear (void) `[static]`

Clear.

Clear the software interrupt

Definition at line 68 of file kernelswi.cpp.

19.13.2.2 void KernelSWI::Config (void) `[static]`

Config.

Configure the software interrupt - must be called before any other software interrupt functions are called.

Definition at line 29 of file kernelswi.cpp.

19.13.2.3 uint8_t KernelSWI::DI ( ) `[static]`

DI.

Disable the SWI flag itself

Returns

previous status of the SWI, prior to the DI call

Definition at line 50 of file kernelswi.cpp.

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

RI.

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.

19.13.2.5 void KernelSWI::Start (void) `[static]`

Start.

Enable ("Start") the software interrupt functionality

Definition at line 37 of file kernelswi.cpp.

19.13.2.6 void KernelSWI::Stop (void) `[static]`

Stop.

Disable the software interrupt functionality

Definition at line 44 of file kernelswi.cpp.

19.13.2.7   void KernelSWI::Trigger (void) `[static]`

Trigger.

Call the software interrupt

Definition at line 74 of file kernelswi.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelswi.h
- /home/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelswi.cpp

19.14   KernelTimer Class Reference

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

`#include <kerneltimer.h>`

Static Public Member Functions

- static void Config (void)

    Config.
- static void Start (void)

    Start.
- static void Stop (void)

    Stop.
- static uint8_t DI (void)

    DI.
- static void RI (bool bEnable_)

    RI.
- static void EI (void)

    EI.
- static PORT_TIMER_COUNT_TYPE SubtractExpiry (PORT_TIMER_COUNT_TYPE uInterval_)

    SubtractExpiry.
- static PORT_TIMER_COUNT_TYPE TimeToExpiry (void)

    TimeToExpiry.
- static PORT_TIMER_COUNT_TYPE SetExpiry (uint32_t u32Interval_)

    SetExpiry.
- static PORT_TIMER_COUNT_TYPE GetOvertime (void)

    GetOvertime.
- static void ClearExpiry (void)

    ClearExpiry.
- static PORT_TIMER_COUNT_TYPE Read (void)

    Read.

19.14.1   Detailed Description

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

Definition at line 31 of file kerneltimer.h.

19.14.2 Member Function Documentation

19.14.2.1 void KernelTimer::ClearExpiry (void) `[static]`

ClearExpiry.

Clear the hardware timer expiry register

Definition at line 136 of file kerneltimer.cpp.

19.14.2.2 void KernelTimer::Config (void) `[static]`

Config.

Initializes the kernel timer before use

Definition at line 33 of file kerneltimer.cpp.

19.14.2.3 uint8_t KernelTimer::DI (void) `[static]`

DI.

Disable the kernel timer's expiry interrupt

Definition at line 144 of file kerneltimer.cpp.

19.14.2.4 void KernelTimer::EI (void) `[static]`

EI.

Enable the kernel timer's expiry interrupt

Definition at line 157 of file kerneltimer.cpp.

19.14.2.5 PORT_TIMER_COUNT_TYPE KernelTimer::GetOvertime (void) `[static]`

GetOvertime.

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 112 of file kerneltimer.cpp.

19.14.2.6 PORT_TIMER_COUNT_TYPE KernelTimer::Read (void) `[static]`

Read.

Safely read the current value in the timer register

Returns

> Value held in the timer register

Examples:

> lab9_dynamic_threads/main.cpp.

Definition at line 66 of file kerneltimer.cpp.

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

RI.

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

Parameters

*bEnable_*	1 enable, 0 disable

Definition at line 163 of file kerneltimer.cpp.

19.14.2.8 PORT_TIMER_COUNT_TYPE KernelTimer::SetExpiry (uint32_t *u32Interval_*) `[static]`

SetExpiry.

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 118 of file kerneltimer.cpp.

19.14.2.9 void KernelTimer::Start (void) `[static]`

Start.

Starts the kernel time (must be configured first)

Definition at line 39 of file kerneltimer.cpp.

19.14.2.10 void KernelTimer::Stop (void) `[static]`

Stop.

Shut down the kernel timer, used when no timers are scheduled

Definition at line 54 of file kerneltimer.cpp.

19.14.2.11 PORT_TIMER_COUNT_TYPE KernelTimer::SubtractExpiry (PORT_TIMER_COUNT_TYPE *uInterval_*) `[static]`

SubtractExpiry.

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.

19.14.2.12 PORT_TIMER_COUNT_TYPE KernelTimer::TimeToExpiry (void) `[static]`

TimeToExpiry.

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/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h
- /home/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kerneltimer.cpp

19.15 LinkList Class Reference

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

```
#include <ll.h>
```

Inheritance diagram for LinkList:
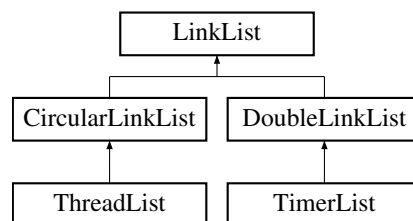
Public Member Functions

- void Init ()

    Init.
- LinkListNode ∗ GetHead ()

    GetHead.
- LinkListNode ∗ GetTail ()

    GetTail.

Protected Attributes

- LinkListNode ∗ m_pstHead

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

    Pointer to the tail node in the list.

19.15.1 Detailed Description

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

Definition at line 109 of file ll.h.

19.15.2   Member Function Documentation

19.15.2.1   LinkListNode∗ **LinkList::GetHead ( )** `[inline]`

GetHead.

Get the head node in the linked list

Returns

Pointer to the head node in the list

Definition at line 134 of file ll.h.

19.15.2.2   LinkListNode∗ **LinkList::GetTail ( )** `[inline]`

GetTail.

Get the tail node of the linked list

Returns

Pointer to the tail node in the list

Definition at line 142 of file ll.h.

19.15.2.3   void LinkList::Init (void) `[inline]`

Init.

Clear the linked list.

Definition at line 121 of file ll.h.

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

- /home/moslevin/mark3-source/embedded/kernel/public/ll.h

19.16   LinkListNode Class Reference

Basic linked-list node data structure.

`#include <ll.h>`

Inheritance diagram for LinkListNode:
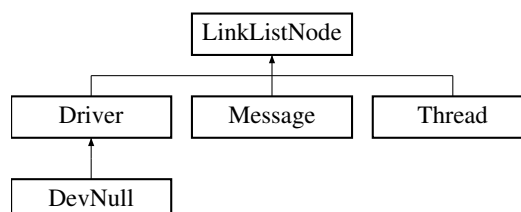
Public Member Functions

- LinkListNode ∗ GetNext (void)

GetNext.

- LinkListNode ∗ GetPrev (void)

    GetPrev.

Protected Member Functions

- void ClearNode ()

    ClearNode.

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**
- class **ThreadList**

19.16.1 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.

19.16.2 Member Function Documentation

19.16.2.1 void LinkListNode::ClearNode ( ) `[protected]`

ClearNode.

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

Definition at line 40 of file ll.cpp.

19.16.2.2 **LinkListNode**∗ **LinkListNode::GetNext (void)** `[inline]`

GetNext.

Returns a pointer to the next node in the list.

Returns

    a pointer to the next node in the list.

Definition at line 90 of file ll.h.

19.16.2.3 LinkListNode∗ LinkListNode::GetPrev (void) `[inline]`

GetPrev.

Returns a pointer to the previous node in the list.

Returns

a pointer to the previous node in the list.

Definition at line 98 of file ll.h.

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

- /home/moslevin/mark3-source/embedded/kernel/public/ll.h
- /home/moslevin/mark3-source/embedded/kernel/ll.cpp

19.17 Mailbox Class Reference

The Mailbox class implements an IPC mechnism based on envelopes containing data of a fixed size (configured at initialization) that reside within a buffer of memory provided by the user.

```
#include <mailbox.h>
```

Public Member Functions

- void Init (void ∗pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_)

    Init.
- bool Send (void ∗pvData_)

    Send.
- bool SendTail (void ∗pvData_)

    SendTail.
- bool Send (void ∗pvData_, uint32_t u32TimeoutMS_)

    Send.
- bool SendTail (void ∗pvData_, uint32_t u32TimeoutMS_)

    SendTail.
- void Receive (void ∗pvData_)

    Receive.
- void ReceiveTail (void ∗pvData_)

    ReceiveTail.
- bool Receive (void ∗pvData_, uint32_t u32TimeoutMS_)

    Receive.
- bool ReceiveTail (void ∗pvData_, uint32_t u32TimeoutMS_)

    ReceiveTail.

Private Member Functions

- void ∗ GetHeadPointer (void)

    GetHeadPointer.
- void ∗ GetTailPointer (void)

    GetTailPointer.
- void CopyData (const void ∗src_, const void ∗dst_, uint16_t len_)

    CopyData.

- void MoveTailForward (void)

    MoveTailForward.
- void MoveHeadForward (void)

    MoveHeadForward.
- void MoveTailBackward (void)

    MoveTailBackward.
- void MoveHeadBackward (void)

    MoveHeadBackward.
- bool Send_i (const void ∗pvData_, bool bTail_, uint32_t u32WaitTimeMS_)

    Send_i.
- bool Receive_i (const void ∗pvData_, bool bTail_, uint32_t u32WaitTimeMS_)

    Receive_i.

Private Attributes

- uint16_t m_u16Head

    Current head index.
- uint16_t m_u16Tail

    Current tail index.
- uint16_t m_u16Count

    Count of items in the mailbox.
- volatile uint16_t m_u16Free

    Current number of free slots in the mailbox.
- uint16_t m_u16ElementSize

    Size of the objects tracked in this mailbox.
- const void ∗ m_pvBuffer

    Pointer to the data-buffer managed by this mailbox.
- Semaphore m_clRecvSem

    Counting semaphore used to synchronize threads on the object.
- Semaphore m_clSendSem

    Binary semaphore for send-blocked threads.

19.17.1 Detailed Description

The Mailbox class implements an IPC mechnism based on envelopes containing data of a fixed size (configured at initialization) that reside within a buffer of memory provided by the user.

Examples:

lab11_mailboxes/main.cpp.

Definition at line 35 of file mailbox.h.

19.17.2 Member Function Documentation

19.17.2.1 void Mailbox::CopyData (const void ∗ *src_,* const void ∗ *dst_,* uint16_t *len_*) `[inline],[private]`

CopyData.

Perform a direct byte-copy from a source to a destination object.

Parameters

*src_*	Pointer to an object to read from
*dst_*	Pointer to an object to write to
*len_*	Length to copy (in bytes)

Definition at line 238 of file mailbox.h.

19.17.2.2 void∗ Mailbox::GetHeadPointer (void) `[inline],[private]`

GetHeadPointer.

Return a pointer to the current head of the mailbox's internal circular buffer.

Returns

pointer to the head element in the mailbox

Definition at line 207 of file mailbox.h.

19.17.2.3 void∗ Mailbox::GetTailPointer (void) `[inline],[private]`

GetTailPointer.

Return a pointer to the current tail of the mailbox's internal circular buffer.

Returns

pointer to the tail element in the mailbox

Definition at line 222 of file mailbox.h.

19.17.2.4 void Mailbox::Init (void ∗ *pvBuffer_,* uint16_t *u16BufferSize_,* uint16_t *u16ElementSize_*)

Init.

Initialize the mailbox object prior to its use. This must be called before any calls can be made to the object.

Parameters

*pvBuffer_*	Pointer to the static buffer to use for the mailbox
u16BufferSize↵ _	Size of the mailbox buffer, in bytes
*u16Element↵ Size_*	Size of each envelope, in bytes

Examples:

lab11_mailboxes/main.cpp.

Definition at line 51 of file mailbox.cpp.

19.17.2.5 void Mailbox::MoveHeadBackward (void) `[inline],[private]`

MoveHeadBackward.

Move the head index backward one element

Definition at line 291 of file mailbox.h.

19.17.2.6   void Mailbox::MoveHeadForward (void) `[inline],[private]`

MoveHeadForward.

Move the head index forward one element

Definition at line 265 of file mailbox.h.

19.17.2.7   void Mailbox::MoveTailBackward (void) `[inline],[private]`

MoveTailBackward.

Move the tail index backward one element

Definition at line 278 of file mailbox.h.

19.17.2.8   void Mailbox::MoveTailForward (void) `[inline],[private]`

MoveTailForward.

Move the tail index forward one element

Definition at line 252 of file mailbox.h.

19.17.2.9   void Mailbox::Receive (void ∗ _pvData_)

Receive.

Read one envelope from the head of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered.

Parameters

*pvData_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.

Examples:

> lab11_mailboxes/main.cpp.

Definition at line 89 of file mailbox.cpp.

19.17.2.10   bool Mailbox::Receive (void ∗ _pvData_, uint32_t _u32TimeoutMS_)

Receive.

Read one envelope from the head of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered, or the specified time has elapsed without delivery.

Parameters

*pvData_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
*u32TimeoutM↩ S_*	Maximum time to wait for delivery.

Returns

> true - envelope was delivered, false - delivery timed out.

Definition at line 102 of file mailbox.cpp.

19.17.2.11   bool Mailbox::Receive_i (const void ∗ *pvData_,* bool *bTail_,* uint32_t *u32WaitTimeMS_*)   `[private]`

Receive_i.

Internal method which implements all Read() methods in the class.

Parameters

*pvData_*	Pointer to the envelope data
*bTail_*	true - read from tail, false - read from head
*u32WaitTimeM↩ S_*	Time to wait before timeout (in ms).

Returns

true - read successfully, false - timeout.

Definition at line 244 of file mailbox.cpp.

19.17.2.12   void Mailbox::ReceiveTail (void ∗ *pvData_*)

ReceiveTail.

Read one envelope from the tail of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered.

Parameters

*pvData_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.

Definition at line 110 of file mailbox.cpp.

19.17.2.13   bool Mailbox::ReceiveTail (void ∗ *pvData_,* uint32_t *u32TimeoutMS_*)

ReceiveTail.

Read one envelope from the tail of the mailbox. If the mailbox is currently empty, the calling thread will block until an envelope is delivered, or the specified time has elapsed without delivery.

Parameters

*pvData_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
*u32TimeoutM↩ S_*	Maximum time to wait for delivery.

Returns

true - envelope was delivered, false - delivery timed out.

Definition at line 123 of file mailbox.cpp.

19.17.2.14   bool Mailbox::Send (void ∗ *pvData_*)

Send.

Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the head of the mailbox.

Parameters

*pvData_*	Pointer to the data object to send to the mailbox.

Returns

> true - envelope was delivered, false - mailbox is full.

Examples:

> lab11_mailboxes/main.cpp.

Definition at line 131 of file mailbox.cpp.

19.17.2.15 bool Mailbox::Send (void ∗ *pvData_,* uint32_t *u32TimeoutMS_*)

Send.

Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the head of the mailbox.

Parameters

*pvData_*	Pointer to the data object to send to the mailbox.
*u32TimeoutM↩ S_*	Maximum time to wait for a free transmit slot

Returns

> true - envelope was delivered, false - mailbox is full.

Definition at line 156 of file mailbox.cpp.

19.17.2.16 bool Mailbox::Send_i (const void ∗ *pvData_,* bool *bTail_,* uint32_t *u32WaitTimeMS_*) `[private]`

Send_i.

Internal method which implements all Send() methods in the class.

Parameters

*pvData_*	Pointer to the envelope data
*bTail_*	true - write to tail, false - write to head
*u32WaitTimeM↩ S_*	Time to wait before timeout (in ms).

Returns

> true - data successfully written, false - buffer full

Definition at line 174 of file mailbox.cpp.

19.17.2.17 bool Mailbox::SendTail (void ∗ *pvData_*)

SendTail.

Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the tail of the mailbox.

Parameters

*pvData_*	Pointer to the data object to send to the mailbox.

Returns

> true - envelope was delivered, false - mailbox is full.

Definition at line 143 of file mailbox.cpp.

19.17.2.18   bool Mailbox::SendTail (void ∗ *pvData_,* uint32_t *u32TimeoutMS_*)

SendTail.

Send an envelope to the mailbox. This safely copies the data contents of the datastructure to the previously-initialized mailbox buffer. If there is a thread already blocking, awaiting delivery to the mailbox, it will be unblocked at this time.

This method delivers the envelope at the tail of the mailbox.

Parameters

*pvData_*	Pointer to the data object to send to the mailbox.
*u32TimeoutM↩ S_*	Maximum time to wait for a free transmit slot

Returns

> true - envelope was delivered, false - mailbox is full.

Definition at line 164 of file mailbox.cpp.

19.17.3   Member Data Documentation

19.17.3.1   Semaphore Mailbox::m_clSendSem `[private]`

Binary semaphore for send-blocked threads.

Definition at line 360 of file mailbox.h.

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

- /home/moslevin/mark3-source/embedded/kernel/public/mailbox.h
- /home/moslevin/mark3-source/embedded/kernel/mailbox.cpp

19.18   Message Class Reference

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

```
#include <message.h>
```

Inheritance diagram for Message:
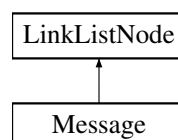
Public Member Functions

- void Init ()

    Init.
- void SetData (void *pvData_)

    SetData.
- void * GetData ()

    GetData.
- void SetCode (uint16_t u16Code_)

    SetCode.
- uint16_t GetCode ()

    GetCode.

Private Attributes

- void * m_pvData

    Pointer to the message data.
- uint16_t m_u16Code

    Message code, providing context for the message.

Additional Inherited Members

19.18.1   Detailed Description

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

Examples:

   lab8_messages/main.cpp.

Definition at line 99 of file message.h.

19.18.2   Member Function Documentation

19.18.2.1   uint16_t Message::GetCode ( ) `[inline]`

GetCode.

Return the code set in the message upon receipt

Returns

   user code set in the object

Examples:

   lab8_messages/main.cpp.

Definition at line 146 of file message.h.

19.18.2.2 **void∗ Message::GetData ( )** `[inline]`

GetData.

Get the data pointer stored in the message upon receipt

Returns

Pointer to the data set in the message object

Examples:

lab8_messages/main.cpp.

Definition at line 130 of file message.h.

19.18.2.3 **void Message::Init (void)** `[inline]`

Init.

Initialize the data and code in the message.

Definition at line 108 of file message.h.

19.18.2.4 **void Message::SetCode (uint16_t *u16Code_*)** `[inline]`

SetCode.

Set the code in the message before transmission

Parameters

*u16Code_*	Data code to set in the object

Examples:

lab8_messages/main.cpp.

Definition at line 138 of file message.h.

19.18.2.5 **void Message::SetData (void ∗ *pvData_*)** `[inline]`

SetData.

Set the data pointer for the message before transmission.

Parameters

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

Examples:

lab8_messages/main.cpp.

Definition at line 122 of file message.h.

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

- /home/moslevin/mark3-source/embedded/kernel/public/message.h

19.19 MessagePool Class Reference

Implements a list of message objects.

```
#include <message.h>
```

Public Member Functions

- void Init ()

    Init.
- void Push (Message ∗pclMessage_)

    Push.
- Message ∗ Pop ()

    Pop.
- Message ∗ GetHead ()

    GetHead.

Private Attributes

- DoubleLinkList m_clList

    Linked list used to manage the Message objects.

19.19.1 Detailed Description

Implements a list of message objects.

Definition at line 159 of file message.h.

19.19.2 Member Function Documentation

19.19.2.1 Message ∗ MessagePool::GetHead ( )

GetHead.

Return a pointer to the first element in the message list

Returns

Definition at line 83 of file message.cpp.

19.19.2.2 void MessagePool::Init (void)

Init.

Initialize the message queue prior to use

Definition at line 50 of file message.cpp.

19.19.2.3 Message ∗ MessagePool::Pop ( )

Pop.

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

Returns

Pointer to a Message object

Definition at line 68 of file message.cpp.

19.19.2.4 void MessagePool::Push (Message ∗ *pclMessage_*)

Push.

Return a previously-claimed message object back to the queue. used once the message has been processed by a receiver.

Parameters

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

Definition at line 56 of file message.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/message.h
- /home/moslevin/mark3-source/embedded/kernel/message.cpp

19.20 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 ()

    Init.
- Message ∗ Receive ()

    Receive.
- Message ∗ Receive (uint32_t u32TimeWaitMS_)

    Receive.
- void Send (Message ∗pclSrc_)

    Send.
- uint16_t GetCount ()

    GetCount.

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.

19.20.1 Detailed Description

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

Examples:

> lab8_messages/main.cpp.

Definition at line 269 of file message.h.

19.20.2 Member Function Documentation

19.20.2.1 uint16_t MessageQueue::GetCount ( )

GetCount.

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

Returns

> Count of pending messages in the queue.

Definition at line 193 of file message.cpp.

19.20.2.2 void MessageQueue::Init (void)

Init.

Initialize the message queue prior to use.

Examples:

> lab8_messages/main.cpp.

Definition at line 124 of file message.cpp.

19.20.2.3 Message ∗ MessageQueue::Receive ( )

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

Examples:

> lab8_messages/main.cpp.

Definition at line 130 of file message.cpp.

19.20.2.4 Message ∗ MessageQueue::Receive (uint32_t *u32TimeWaitMS_*)

Receive.

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

u32TimeWaitM↩ *S_*	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 141 of file message.cpp.

19.20.2.5 Message ∗ **MessageQueue::Receive_i (uint32_t *u32TimeWaitMS_*)** `[private]`

Receive_i.

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

Parameters

u32TimeWaitM↩ *S_*	Time (in ms) to block, 0 for un-timed call.

Returns

Pointer to a message, or 0 on timeout.

Definition at line 149 of file message.cpp.

19.20.2.6 void MessageQueue::Send (Message ∗ *pclSrc_* **)**

Send.

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

Examples:

lab8_messages/main.cpp.

Definition at line 177 of file message.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/message.h
- /home/moslevin/mark3-source/embedded/kernel/message.cpp

19.21 Mutex Class Reference

Mutual-exclusion locks, based on BlockingObject.

```
#include <mutex.h>
```

Inheritance diagram for Mutex:

```
                    BlockingObject

                        Mutex
```
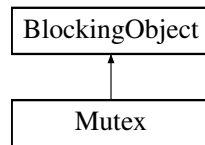
Public Member Functions

- void Init ()

    Init.
- void Claim ()

    Claim.
- bool Claim (uint32_t u32WaitTimeMS_)

    Claim.
- void WakeMe (Thread ∗pclOwner_)

    WakeMe.
- void Release ()

    Release.

Private Member Functions

- uint8_t WakeNext ()

    WakeNext.
- 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

19.21.1   Detailed Description

Mutual-exclusion locks, based on BlockingObject.

Examples:

lab5_mutexes/main.cpp.

Definition at line 68 of file mutex.h.

19.21.2 Member Function Documentation

19.21.2.1 void Mutex::Claim (void)

Claim.

Claim the mutex. When the mutex is claimed, no other thread can claim a region protected by the object. If another Thread currently holds the Mutex when the Claim method is called, that Thread will block until the current owner of the mutex releases the Mutex.

If the calling Thread's priority is lower than that of a Thread that currently owns the Mutex object, then the priority of that Thread will be elevated to that of the highest-priority calling object until the Mutex is released. This property is known as "Priority Inheritence"

Note: A single thread can recursively claim a mutex up to a count of

1. Attempting to claim a mutex beyond that will cause a kernel panic.

Examples:

lab5_mutexes/main.cpp.

Definition at line 215 of file mutex.cpp.

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

Claim.

Claim a mutex, with timeout.

Parameters

*u32WaitTimeM↩ S_*	

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.

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

Claim_i.

Abstracts out timed/non-timed mutex claim operations.

Parameters

*u32WaitTimeM↩ S_*	Time in MS to wait, 0 for infinite

Returns

true on successful claim, false otherwise

Definition at line 120 of file mutex.cpp.

19.21.2.4 void Mutex::Init (void)

Init.

Initialize a mutex object for use - must call this function before using the object.

Examples:

lab5_mutexes/main.cpp.

Definition at line 109 of file mutex.cpp.

19.21.2.5 void Mutex::Release ( )

Release.

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

If there are Threads waiting for the Mutex to become available, then the highest priority Thread will be unblocked at this time and will claim the Mutex lock immediately - this may result in an immediate context switch, depending on relative priorities.

If the calling Thread's priority was boosted as a result of priority inheritence, the Thread's previous priority will also be restored at this time.

Note that if a Mutex is held recursively, it must be Release'd the same number of times that it was Claim'd before it will be availabel for use by another Thread.

Examples:

lab5_mutexes/main.cpp.

Definition at line 233 of file mutex.cpp.

19.21.2.6 void Mutex::WakeMe (Thread ∗ pclOwner_)

WakeMe.

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 79 of file mutex.cpp.

19.21.2.7 uint8_t Mutex::WakeNext ( ) `[private]`

WakeNext.

Wake the next thread waiting on the Mutex.

Definition at line 88 of file mutex.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/mutex.h
- /home/moslevin/mark3-source/embedded/kernel/mutex.cpp

19.22 Notify Class Reference

The Notify class is a blocking object type, that allows one or more threads to wait for an event to occur before resuming operation.

```
#include <notify.h>
```

Inheritance diagram for Notify:

```
┌─────────────────┐
│  BlockingObject  │
└─────────────────┘
         ▲
┌─────────────────┐
│     Notify      │
└─────────────────┘
```
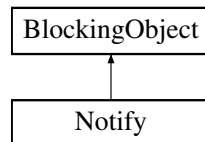
Public Member Functions

- void Init (void)

    Init.
- void Signal (void)

    Signal.
- void Wait (bool ∗pbFlag_)

    Wait.
- bool Wait (uint32_t u32WaitTimeMS_, bool ∗pbFlag_)

    Wait.
- void WakeMe (Thread ∗pclChosenOne_)

    WakeMe.

Additional Inherited Members

19.22.1 Detailed Description

The Notify class is a blocking object type, that allows one or more threads to wait for an event to occur before resuming operation.

Examples:

    lab10_notifications/main.cpp.

Definition at line 33 of file notify.h.

19.22.2 Member Function Documentation

19.22.2.1 void Notify::Init (void)

Init.

Initialze the Notification object prior to use.

Examples:

    lab10_notifications/main.cpp.

Definition at line 67 of file notify.cpp.

19.22.2.2 void Notify::Signal (void)

Signal.

Signal the notification object. This will cause the highest priority thread currently blocking on the object to wake. If no threads are currently blocked on the object, the call has no effect.

Examples:

lab10_notifications/main.cpp.

Definition at line 73 of file notify.cpp.

19.22.2.3 void Notify::Wait (bool ∗ _pbFlag__)

Wait.

Block the current thread, waiting for a signal on the object.

Parameters

_pbFlag__	Flag set to false on block, and true upon wakeup.

Examples:

lab10_notifications/main.cpp.

Definition at line 94 of file notify.cpp.

19.22.2.4 bool Notify::Wait (uint32_t _u32WaitTimeMS__,_ bool ∗ _pbFlag__)

Wait.

Block the current thread, waiting for a signal on the object.

Parameters

_u32WaitTimeM↵ S__	Time to wait for the notification event.
_pbFlag__	Flag set to false on block, and true upon wakeup.

Returns

true on notification, false on timeout

Definition at line 111 of file notify.cpp.

19.22.2.5 void Notify::WakeMe (Thread ∗ _pclChosenOne__)

WakeMe.

Wake the specified thread from its current blocking queue. Note that this is only public in order to be accessible from a timer callack.

Parameters

| *pclChosenOne↩* | Thread to wake up |
| *_* | |

Definition at line 147 of file notify.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/notify.h
- /home/moslevin/mark3-source/embedded/kernel/notify.cpp

19.23 PriorityMap Class Reference

The PriorityMap class.

```
#include <priomap.h>
```

Public Member Functions

- PriorityMap ()

  PriorityMap.
- void Set (PORT_PRIO_TYPE uXPrio_)

  Set Set the priority map bitmap data, at all levels, for the given priority.
- void Clear (PORT_PRIO_TYPE uXPrio_)

  Clear Clear the priority map bitmap data, at all levels, for the given priority.
- PORT_PRIO_TYPE HighestPriority (void)

  HighestPriority.

19.23.1 Detailed Description

The PriorityMap class.

Definition at line 70 of file priomap.h.

19.23.2 Constructor & Destructor Documentation

19.23.2.1 PriorityMap::PriorityMap ( )

PriorityMap.

Initialize the priority map object, clearing the bitamp data to all 0's.

Definition at line 51 of file priomap.cpp.

19.23.3 Member Function Documentation

19.23.3.1 void PriorityMap::Clear (PORT_PRIO_TYPE *uXPrio_*)

Clear Clear the priority map bitmap data, at all levels, for the given priority.

Parameters

*uXPrio_*	Priority level to clear the bitmap data for.

Definition at line 78 of file priomap.cpp.

19.23.3.2 PORT_PRIO_TYPE PriorityMap::HighestPriority (void)

HighestPriority.

Computes the numeric priority of the highest-priority thread represented in the priority map.

Returns

Highest priority ready-thread's number.

Definition at line 94 of file priomap.cpp.

19.23.3.3 void PriorityMap::Set (PORT_PRIO_TYPE *uXPrio_*)

Set Set the priority map bitmap data, at all levels, for the given priority.

Parameters

*uXPrio_*	Priority level to set the bitmap data for.

Definition at line 64 of file priomap.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/priomap.h
- /home/moslevin/mark3-source/embedded/kernel/priomap.cpp

19.24 Profiler Class Reference

System profiling timer interface.

```
#include <kernelprofile.h>
```

Static Public Member Functions

- static void Init ()

    Init.
- static void Start ()

    Start.
- static void Stop ()

    Stop.
- static PORT_TIMER_COUNT_TYPE Read ()

    Read.
- static void Process ()

    Process.
- static uint32_t GetEpoch ()

    GetEpoch.

19.24.1 Detailed Description

System profiling timer interface.

Definition at line 37 of file kernelprofile.h.

19.24.2 Member Function Documentation

19.24.2.1 static uint32_t Profiler::GetEpoch ( ) `[inline],[static]`

GetEpoch.

Return the current timer epoch

Definition at line 81 of file kernelprofile.h.

19.24.2.2 void Profiler::Init (void) `[static]`

Init.

Initialize the global system profiler. Must be called prior to use.

Definition at line 32 of file kernelprofile.cpp.

19.24.2.3 void Profiler::Process (void) `[static]`

Process.

Process the profiling counters from ISR.

Definition at line 70 of file kernelprofile.cpp.

19.24.2.4 PORT_TIMER_COUNT_TYPE Profiler::Read ( ) `[static]`

Read.

Read the current tick count in the timer.

Definition at line 58 of file kernelprofile.cpp.

19.24.2.5 void Profiler::Start (void) `[static]`

Start.

Start the global profiling timer service.

Definition at line 42 of file kernelprofile.cpp.

19.24.2.6 void Profiler::Stop ( ) `[static]`

Stop.

Stop the global profiling timer service

Definition at line 51 of file kernelprofile.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kernelprofile.h
- /home/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/kernelprofile.cpp

19.25 ProfileTimer Class Reference

Profiling timer.

```
#include <profile.h>
```

Public Member Functions

- void Init ()

    Init.
- void Start ()

    Start.
- void Stop ()

    Stop.
- uint32_t GetAverage ()

    GetAverage.
- uint32_t GetCurrent ()

    GetCurrent.

Private Member Functions

- uint32_t ComputeCurrentTicks (uint16_t u16Count_, uint32_t u32Epoch_)

    ComputeCurrentTicks.

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.

19.25.1 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.

19.25.2 Member Function Documentation

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

ComputeCurrentTicks.

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 107 of file profile.cpp.

19.25.2.2   uint32_t ProfileTimer::GetAverage ( )

GetAverage.

Get the average time associated with this operation.

Returns

Average tick count normalized over all iterations

Definition at line 83 of file profile.cpp.

19.25.2.3   uint32_t ProfileTimer::GetCurrent ( )

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 92 of file profile.cpp.

19.25.2.4   void ProfileTimer::Init (void)

Init.

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.

19.25.2.5   void ProfileTimer::Start (void)

Start.

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.

19.25.2.6   void ProfileTimer::Stop ( )

Stop.

Stop the current profiling session, adding to the cumulative time for this timer, and the total iteration count.

Definition at line 65 of file profile.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/profile.h
- /home/moslevin/mark3-source/embedded/kernel/profile.cpp

19.26 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 ()

    UpdateTimer.
- static void AddThread (Thread ∗pclThread_)

    AddThread.
- static void RemoveThread ()

    RemoveThread.
- static void SetInTimer (void)

    SetInTimer.
- static void ClearInTimer (void)

    ClearInTimer.

Static Private Member Functions

- static void SetTimer (Thread ∗pclThread_)

    SetTimer.

19.26.1 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.

19.26.2 Member Function Documentation

19.26.2.1 void Quantum::AddThread (Thread ∗ *pclThread_*) `[static]`

AddThread.

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 86 of file quantum.cpp.

19.26.2.2 static void Quantum::ClearInTimer (void) `[inline],[static]`

ClearInTimer.

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

Definition at line 83 of file quantum.h.

**19.26.2.3 void Quantum::RemoveThread (void) ** `[static]`

RemoveThread.

Remove the thread from the quantum timer. This will cancel the timer.

Definition at line 111 of file quantum.cpp.

**19.26.2.4 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.

**19.26.2.5 void Quantum::SetTimer (Thread ∗ pclThread_) ** `[static],[private]`

SetTimer.

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 76 of file quantum.cpp.

**19.26.2.6 void Quantum::UpdateTimer (void) ** `[static]`

UpdateTimer.

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 123 of file quantum.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/quantum.h
- /home/moslevin/mark3-source/embedded/kernel/quantum.cpp

19.27 Scheduler Class Reference

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

```
#include <scheduler.h>
```

Static Public Member Functions

- static void Init ()

    Init.
- static void Schedule ()

    Schedule.
- static void Add (Thread ∗pclThread_)

Add.

- static void Remove (Thread *pclThread_)

    Remove.

- static bool SetScheduler (bool bEnable_)

    SetScheduler.

- static Thread * GetCurrentThread ()

    GetCurrentThread.

- static volatile Thread * GetNextThread ()

    GetNextThread.

- static ThreadList * GetThreadList (PORT_PRIO_TYPE uXPriority_)

    GetThreadList.

- static ThreadList * GetStopList ()

    GetStopList.

- static bool IsEnabled ()

    IsEnabled.

- 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 [KERNEL_NUM_PRIORITIES]

    ThreadLists for all threads at all priorities.

- static PriorityMap m_clPrioMap

    Priority bitmap lookup structure, 1-bit per thread priority.

19.27.1 Detailed Description

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

Definition at line 62 of file scheduler.h.

19.27.2 Member Function Documentation

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

Add.

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 89 of file scheduler.cpp.

19.27.2.2   static Thread∗ Scheduler::GetCurrentThread ( ) `[inline],[static]`

GetCurrentThread.

Return the pointer to the currently-running thread.

Returns

Pointer to the currently-running thread

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 121 of file scheduler.h.

19.27.2.3   static volatile Thread∗ Scheduler::GetNextThread ( ) `[inline],[static]`

GetNextThread.

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 130 of file scheduler.h.

19.27.2.4   static ThreadList∗ Scheduler::GetStopList ( ) `[inline],[static]`

GetStopList.

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 150 of file scheduler.h.

19.27.2.5   static ThreadList∗ Scheduler::GetThreadList (PORT_PRIO_TYPE *uXPriority_*) `[inline],[static]`

GetThreadList.

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

Parameters

*uXPriority_*	Priority level of the threadlist

Returns

Pointer to the ThreadList for the given priority level

Definition at line 141 of file scheduler.h.

19.27.2.6   void Scheduler::Init (void) `[static]`

Init.

Intiailize the scheduler, must be called before use.

Definition at line 54 of file scheduler.cpp.

19.27.2.7 **static bool Scheduler::IsEnabled ( )** `[inline],[static]`

IsEnabled.

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

Returns

true - scheduler enabled, false - disabled

Definition at line 159 of file scheduler.h.

19.27.2.8 **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 166 of file scheduler.h.

19.27.2.9 **void Scheduler::Remove (Thread ∗ *pclThread_*)** `[static]`

Remove.

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 95 of file scheduler.cpp.

19.27.2.10 **void Scheduler::Schedule ( )** `[static]`

Schedule.

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 63 of file scheduler.cpp.

19.27.2.11 **bool Scheduler::SetScheduler (bool *bEnable_*)** `[static]`

SetScheduler.

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 101 of file scheduler.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/scheduler.h
- /home/moslevin/mark3-source/embedded/kernel/scheduler.cpp

19.28 Semaphore Class Reference

Binary & Counting semaphores, based on BlockingObject base class.

```
#include <ksemaphore.h>
```

Inheritance diagram for Semaphore:

```
┌─────────────────┐
│ BlockingObject  │
└─────────────────┘
         ▲
         │
┌─────────────────┐
│   Semaphore     │
└─────────────────┘
```
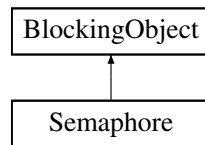
Public Member Functions

- 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

19.28.1 Detailed Description

Binary & Counting semaphores, based on BlockingObject base class.

Examples:

buffalogger/main.cpp, lab4_semaphores/main.cpp, lab6_timers/main.cpp, and lab9_dynamic_threads/main.↵
cpp.

Definition at line 37 of file ksemaphore.h.

19.28.2 Member Function Documentation

19.28.2.1 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 234 of file ksemaphore.cpp.

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

Initialize a semaphore before use.

Must be called before attempting post/pend operations on the object.

This initialization is required to configure the behavior of the semaphore with regards to the initial and maximum values held by the semaphore. By providing access to the raw initial and maximum count elements of the semaphore, these objects are able to be used as either counting or binary semaphores.

To configure a semaphore object for use as a binary semaphore, set values of 0 and 1 respectively for the initial/maximum value parameters.

Any other combination of values can be used to implement a counting semaphore.

Parameters

*u16InitVal_*	Initial value held by the semaphore
*u16MaxVal_*	Maximum value for the semaphore. Must be nonzero.

Examples:

buffalogger/main.cpp, lab4_semaphores/main.cpp, lab6_timers/main.cpp, and lab9_dynamic_threads/main.↩ cpp.

Definition at line 108 of file ksemaphore.cpp.

19.28.2.3 void Semaphore::Pend ( )

Decrement the semaphore count.

If the count is zero, the calling Thread will block until the semaphore is posted, and the Thread's priority is higher than that of any other Thread blocked on the object.

Examples:

buffalogger/main.cpp, lab4_semaphores/main.cpp, lab6_timers/main.cpp, and lab9_dynamic_threads/main.↩ cpp.

Definition at line 216 of file ksemaphore.cpp.

19.28.2.4   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 227 of file ksemaphore.cpp.

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

Pend_i.

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

Parameters

| *u32WaitTimeM↩* | Time in MS to wait |
| *S_* | |

Returns

true on success, false on failure.

Definition at line 165 of file ksemaphore.cpp.

19.28.2.6   bool Semaphore::Post (   )

Increment the semaphore count.

If the semaphore count is zero at the time this is called, and there are threads blocked on the object, this will immediately unblock the highest-priority blocked Thread.

Note that if the priority of that Thread is higher than the current thread's priority, a context switch will occur and control will be relinquished to that Thread.

Returns

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

Examples:

buffalogger/main.cpp, lab4_semaphores/main.cpp, lab6_timers/main.cpp, and lab9_dynamic_threads/main.↩cpp.

Definition at line 120 of file ksemaphore.cpp.

19.28.2.7   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 82 of file ksemaphore.cpp.

19.28.2.8 uint8_t Semaphore::WakeNext ( ) `[private]`

Wake the next thread waiting on the semaphore.

Used internally.

Definition at line 91 of file ksemaphore.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/ksemaphore.h
- /home/moslevin/mark3-source/embedded/kernel/ksemaphore.cpp

19.29 Thread Class Reference

Object providing fundamental multitasking support in the kernel.

`#include <thread.h>`

Inheritance diagram for Thread:
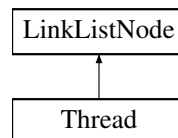
Public Member Functions

- void Init (K_WORD ∗pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_, ThreadEntry_↩
  t pfEntryPoint_, void ∗pvArg_)

    Init.
- void Start ()

    Start.
- void Stop ()

    Stop.
- ThreadList ∗ GetOwner (void)

    GetOwner.
- ThreadList ∗ GetCurrent (void)

    GetCurrent.
- PORT_PRIO_TYPE GetPriority (void)

    GetPriority.
- PORT_PRIO_TYPE GetCurPriority (void)

    GetCurPriority.
- void SetQuantum (uint16_t u16Quantum_)

    SetQuantum.
- uint16_t GetQuantum (void)

    GetQuantum.
- void SetCurrent (ThreadList ∗pclNewList_)

    SetCurrent.
- void SetOwner (ThreadList ∗pclNewList_)

    SetOwner.
- void SetPriority (PORT_PRIO_TYPE uXPriority_)

    SetPriority.

- void InheritPriority (PORT_PRIO_TYPE uXPriority_)

    InheritPriority.
- void Exit ()

    Exit.
- void SetID (uint8_t u8ID_)

    SetID.
- uint8_t GetID ()

    GetID.
- uint16_t GetStackSlack ()

    GetStackSlack.
- 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_)

    Sleep.
- static void USleep (uint32_t u32TimeUs_)

    USleep.
- static void Yield (void)

    Yield.

Private Member Functions

- void SetPriorityBase (PORT_PRIO_TYPE uXPriority_)

    SetPriorityBase.

Static Private Member Functions

- static void ContextSwitchSWI (void)

    ContextSwitchSWI.

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.
- PORT_PRIO_TYPE m_uXPriority

    Default priority of the thread.
- PORT_PRIO_TYPE m_uXCurPriority

    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

19.29.1   Detailed Description

Object providing fundamental multitasking support in the kernel.

Examples:

buffalogger/main.cpp, lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_idle_function/main.cpp, lab3_round_robin/main.cpp, lab4_semaphores/main.cpp, lab5_mutexes/main.↩ cpp, lab6_timers/main.cpp, lab7_events/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_↩ threads/main.cpp.

Definition at line 60 of file thread.h.

19.29.2 Member Function Documentation

19.29.2.1 void Thread::ContextSwitchSWI (void) `[static],[private]`

ContextSwitchSWI.

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 441 of file thread.cpp.

19.29.2.2 void Thread::Exit ( )

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.

Examples:

> lab9_dynamic_threads/main.cpp.

Definition at line 217 of file thread.cpp.

19.29.2.3 PORT_PRIO_TYPE Thread::GetCurPriority (void) `[inline]`

GetCurPriority.

Return the priority of the current thread

Returns

> Priority of the current thread

Definition at line 176 of file thread.h.

19.29.2.4 ThreadList∗ Thread::GetCurrent (void) `[inline]`

GetCurrent.

Return the ThreadList where the thread is currently located

Returns

> Pointer to the thread's current list

Definition at line 159 of file thread.h.

19.29.2.5 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 321 of file thread.h.

19.29.2.6 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 337 of file thread.h.

19.29.2.7 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 485 of file thread.cpp.

19.29.2.8 uint8_t Thread::GetID ( ) `[inline]`

GetID.

Return the 8-bit ID corresponding to this thread.

Returns

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

Definition at line 298 of file thread.h.

19.29.2.9 ThreadList∗ Thread::GetOwner (void) `[inline]`

GetOwner.

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 151 of file thread.h.

19.29.2.10 PORT_PRIO_TYPE Thread::GetPriority (void) `[inline]`

GetPriority.

Return the priority of the current thread

Returns

Priority of the current thread

Definition at line 168 of file thread.h.

19.29.2.11 uint16_t Thread::GetQuantum (void) `[inline]`

GetQuantum.

Get the thread's round-robin execution quantum.

Returns

The thread's quantum

Definition at line 193 of file thread.h.

19.29.2.12 uint16_t Thread::GetStackSlack ( )

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 : Reverse the logic for MCUs where stack grows UP instead of down

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 329 of file thread.cpp.

19.29.2.13 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

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 381 of file thread.h.

19.29.2.14 void Thread::InheritPriority (PORT_PRIO_TYPE *uXPriority_*)

InheritPriority.

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

*uXPriority_*	New Priority to boost to.

Definition at line 434 of file thread.cpp.

19.29.2.15 void Thread::Init (K_WORD ∗ *pwStack_,* uint16_t *u16StackSize_,* PORT_PRIO_TYPE *uXPriority_,* ThreadEntry_t *pfEntryPoint_,* void ∗ *pvArg_*)

Init.

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

*pwStack_*	Pointer to the stack to use for the thread
*u16StackSize_*	Size of the stack (in bytes)
*uXPriority_*	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.

Examples:

buffalogger/main.cpp, lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_idle_function/main.cpp, lab3_round_robin/main.cpp, lab4_semaphores/main.cpp, lab5_mutexes/main.↩ cpp, lab6_timers/main.cpp, lab7_events/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_↩ threads/main.cpp.

Definition at line 70 of file thread.cpp.

19.29.2.16 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 493 of file thread.cpp.

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

SetCurrent.

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.

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

SetEventFlagMask Sets the active event flag bitfield mask.

Parameters

*u16Mask_*	

Definition at line 326 of file thread.h.

19.29.2.19   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 332 of file thread.h.

19.29.2.20   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 479 of file thread.cpp.

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

SetID.

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

Parameters

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

Definition at line 290 of file thread.h.

19.29.2.22   void Thread::SetOwner (ThreadList ∗ *pclNewList_*)   `[inline]`

SetOwner.

Set the thread's owner to the specified thread list

Parameters

*pclNewList_*	Pointer to the threadlist to apply thread ownership

Definition at line 211 of file thread.h.

19.29.2.23   void Thread::SetPriority (PORT_PRIO_TYPE *uXPriority_*)

SetPriority.

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

*uXPriority_*	New priority of the thread

Definition at line 395 of file thread.cpp.

19.29.2.24 void Thread::SetPriorityBase (PORT_PRIO_TYPE *uXPriority_*) `[private]`

SetPriorityBase.

Parameters

*uXPriority_*	

Definition at line 385 of file thread.cpp.

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

SetQuantum.

Set the thread's round-robin execution quantum.

Parameters

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

Examples:

> lab3_round_robin/main.cpp.

Definition at line 185 of file thread.h.

19.29.2.26 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 389 of file thread.h.

19.29.2.27 void Thread::Sleep (uint32_t *u32TimeMs_*) `[static]`

Sleep.

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)

Examples:

> buffalogger/main.cpp, lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.↩
> cpp, lab2_idle_function/main.cpp, lab7_events/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_↩
> threads/main.cpp.

Definition at line 284 of file thread.cpp.

19.29.2.28 void Thread::Start (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.

Examples:

buffalogger/main.cpp, lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_idle_function/main.cpp, lab3_round_robin/main.cpp, lab4_semaphores/main.cpp, lab5_mutexes/main.↵ cpp, lab6_timers/main.cpp, lab7_events/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_↵ threads/main.cpp.

Definition at line 145 of file thread.cpp.

19.29.2.29 void Thread::Stop ( )

Stop.

Stop a thread that's actively scheduled without destroying its stacks. Stopped threads can be restarted using the Start() API.

Definition at line 177 of file thread.cpp.

19.29.2.30 void Thread::USleep (uint32_t *u32TimeUs_*) `[static]`

USleep.

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 306 of file thread.cpp.

19.29.2.31 void Thread::Yield (void) `[static]`

Yield.

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 360 of file thread.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/thread.h
- /home/moslevin/mark3-source/embedded/kernel/thread.cpp

19.30 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:
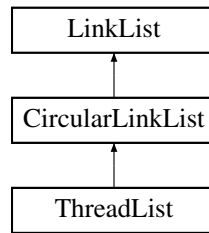
Public Member Functions

- ThreadList ()

    ThreadList.
- void SetPriority (PORT_PRIO_TYPE uXPriority_)

    SetPriority.
- void SetMapPointer (PriorityMap ∗pclMap_)

    SetMapPointer.
- void Add (LinkListNode ∗node_)

    Add.
- void Add (LinkListNode ∗node_, PriorityMap ∗pclMap_, PORT_PRIO_TYPE uXPriority_)

    Add.
- void AddPriority (LinkListNode ∗node_)

    AddPriority.
- void Remove (LinkListNode ∗node_)

    Remove.
- Thread ∗ HighestWaiter ()

    HighestWaiter.

Private Attributes

- PORT_PRIO_TYPE m_uXPriority

    Priority of the threadlist.
- PriorityMap ∗ m_pclMap

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

Additional Inherited Members

19.30.1 Detailed Description

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

Definition at line 35 of file threadlist.h.

19.30.2 Constructor & Destructor Documentation

19.30.2.1 ThreadList::ThreadList ( ) [inline]

ThreadList.

Default constructor - zero-initializes the data.

Definition at line 44 of file threadlist.h.

19.30.3   Member Function Documentation

19.30.3.1   void ThreadList::Add (LinkListNode ∗ *node_*)

Add.

Add a thread to the threadlist.

Parameters

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

Definition at line 52 of file threadlist.cpp.

19.30.3.2   void ThreadList::Add (LinkListNode ∗ *node_,* PriorityMap ∗ *pclMap_,* PORT_PRIO_TYPE *uXPriority_*)

Add.

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

Definition at line 101 of file threadlist.cpp.

19.30.3.3   void ThreadList::AddPriority (LinkListNode ∗ *node_*)

AddPriority.

Add a thread to the list such that threads are ordered from highest to lowest priority from the head of the list.

Parameters

*node_*	Pointer to a thread to add to the list.

Definition at line 65 of file threadlist.cpp.

19.30.3.4   Thread ∗ ThreadList::HighestWaiter ( )

HighestWaiter.

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

Returns

Pointer to the highest-priority thread

Definition at line 124 of file threadlist.cpp.

19.30.3.5   void ThreadList::Remove (LinkListNode ∗ *node_*)

Remove.

Remove the specified thread from the threadlist

Parameters

*node_*	Pointer to the thread to remove

Definition at line 111 of file threadlist.cpp.

19.30.3.6   void ThreadList::SetMapPointer (PriorityMap ∗ *pclMap_*)

SetMapPointer.

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

*pclMap_*	Pointer to the priority map object used to track this thread.

Definition at line 46 of file threadlist.cpp.

19.30.3.7   void ThreadList::SetPriority (PORT_PRIO_TYPE *uXPriority_*)

SetPriority.

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

Parameters

*uXPriority_*	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/moslevin/mark3-source/embedded/kernel/public/threadlist.h
- /home/moslevin/mark3-source/embedded/kernel/threadlist.cpp

19.31   ThreadPort Class Reference

Class defining the architecture specific functions required by the kernel.

```
#include <threadport.h>
```

Static Public Member Functions

- static void StartThreads ()

    StartThreads.

Static Private Member Functions

- static void InitStack (Thread ∗pstThread_)

    InitStack.

Friends

- class **Thread**

19.31.1 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 275 of file threadport.h.

19.31.2 Member Function Documentation

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

InitStack.

Initialize the thread's stack.

Parameters

*pstThread_*	Pointer to the thread to initialize

Definition at line 39 of file threadport.cpp.

19.31.2.2 void ThreadPort::StartThreads ( ) `[static]`

StartThreads.

Function to start the scheduler, initial threads, etc.

Definition at line 130 of file threadport.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/threadport.h
- /home/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/threadport.cpp

19.32 TimerList Class Reference

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

```
#include <timerlist.h>
```

Inheritance diagram for TimerList:
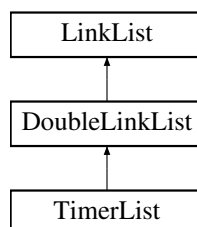
Public Member Functions

- void Init ()
    Init.
- void Add (Timer *pclListNode_)
    Add.

- void Remove (Timer ∗pclListNode_)

    Remove.
- void Process ()

    Process.

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

19.32.1 Detailed Description

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

Definition at line 37 of file timerlist.h.

19.32.2 Member Function Documentation

19.32.2.1 void TimerList::Add (Timer ∗ *pclListNode_*)

Add.

Add a timer to the TimerList.

Parameters

*pclListNode_*	Pointer to the Timer to Add

Definition at line 56 of file timerlist.cpp.

19.32.2.2 void TimerList::Init (void)

Init.

Initialize the TimerList object. Must be called before using the object.

Definition at line 49 of file timerlist.cpp.

19.32.2.3 void TimerList::Process (void)

Process.

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 119 of file timerlist.cpp.

19.32.2.4 void TimerList::Remove (Timer ∗ *pclListNode_*)

Remove.

Remove a timer from the TimerList, cancelling its expiry.

Parameters

*pclListNode_*	Pointer to the Timer to remove

Definition at line 102 of file timerlist.cpp.

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

- /home/moslevin/mark3-source/embedded/kernel/public/timerlist.h
- /home/moslevin/mark3-source/embedded/kernel/timerlist.cpp

19.33 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 ()

    Init.
- static void Add (Timer *pclListNode_)

    Add.
- static void Remove (Timer *pclListNode_)

    Remove.
- static void Process ()

    Process.

Static Private Attributes

- static TimerList m_clTimerList

    TimerList object manipu32ated by the Timer Scheduler.

19.33.1 Detailed Description

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

Definition at line 38 of file timerscheduler.h.

19.33.2 Member Function Documentation

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

Add.

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 56 of file timerscheduler.h.

19.33.2.2 static void TimerScheduler::Init (void) `[inline],[static]`

Init.

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

Definition at line 47 of file timerscheduler.h.

19.33.2.3 static void TimerScheduler::Process (void) `[inline],[static]`

Process.

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 74 of file timerscheduler.h.

19.33.2.4 static void TimerScheduler::Remove (Timer ∗ *pclListNode_*) `[inline],[static]`

Remove.

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 65 of file timerscheduler.h.

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

- /home/moslevin/mark3-source/embedded/kernel/public/timerscheduler.h
- /home/moslevin/mark3-source/embedded/kernel/timerlist.cpp

Chapter 20

File Documentation

20.1 /home/moslevin/mark3-source/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"
```

20.1.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.cpp.

20.2 atomic.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____        _____
00003   ___|    _|__  __|_    |__    |__  _|__  __|_    |__  _____
00004  |    \  /  |  | ||    \       ||    |      ||    |/ /       ||___       |
00005  |     \/   |  | ||     \      ||    |      ||    |\   \     ||__        |
00006  |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]---------------------------------------------
00010
00011 Copyright (c) 2012-2016 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         *pbLock_ = 1;
00144     }
00145     CS_EXIT();
00146     return u8Ret;
00147 }
00148
00149 #endif // KERNEL_USE_ATOMIC
```

20.3 /home/moslevin/mark3-source/embedded/kernel/autoalloc.cpp File Reference

Automatic memory allocation for kernel objects.

```
#include "mark3cfg.h"
#include "mark3.h"
#include "autoalloc.h"
#include "threadport.h"
#include "kernel.h"
```

20.3.1 Detailed Description

Automatic memory allocation for kernel objects.

Definition in file autoalloc.cpp.

20.4 autoalloc.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /    | ||    \      ||    |      ||    |/ /      ||___    |
00005 |     \/     | ||     \     ||     \     ||    |  \      ||___    |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  __||_____|
00007     |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00020 #include "mark3cfg.h"
00021 #include "mark3.h"
00022 #include "autoalloc.h"
00023 #include "threadport.h"
00024 #include "kernel.h"
00025
00026 #if KERNEL_USE_AUTO_ALLOC
00027
00028 // Align to nearest word boundary
00029 #define ALLOC_ALIGN(x) (((x) + (sizeof(K_ADDR) - 1)) & (sizeof(K_ADDR) - 1))
00030
00031 //---------------------------------------------------------------------------
00032 uint8_t AutoAlloc::m_au8AutoHeap[AUTO_ALLOC_SIZE];
00033 K_ADDR  AutoAlloc::m_aHeapTop;
```

```
00034
00035  //---------------------------------------------------------------------------
00036  void AutoAlloc::Init(void)
00037  {
00038      m_aHeapTop = (K_ADDR)(m_au8AutoHeap);
00039  }
00040
00041  //---------------------------------------------------------------------------
00042  void* AutoAlloc::Allocate(uint16_t u16Size_)
00043  {
00044      void* pvRet = 0;
00045
00046      CS_ENTER();
00047      uint16_t u16AllocSize = ALLOC_ALIGN(u16Size_);
00048      if ((((K_ADDR)m_aHeapTop - (K_ADDR)&m_au8AutoHeap[0]) + u16AllocSize) < AUTO_ALLOC_SIZE) {
00049          pvRet = (void*)m_aHeapTop;
00050          m_aHeapTop += u16AllocSize;
00051      }
00052      CS_EXIT();
00053
00054      if (!pvRet) {
00055          Kernel::Panic(PANIC_AUTO_HEAP_EXHAUSTED);
00056      }
00057
00058      return pvRet;
00059  }
00060
00061  #if KERNEL_USE_SEMAPHORE
00062  //---------------------------------------------------------------------------
00063  Semaphore* AutoAlloc::NewSemaphore(void)
00064  {
00065      void* pvObj = Allocate(sizeof(Semaphore));
00066      if (pvObj) {
00067          return new (pvObj) Semaphore();
00068      }
00069      return 0;
00070  }
00071  #endif
00072
00073  #if KERNEL_USE_MUTEX
00074  //---------------------------------------------------------------------------
00075  Mutex* AutoAlloc::NewMutex(void)
00076  {
00077      void* pvObj = Allocate(sizeof(Mutex));
00078      if (pvObj) {
00079          return new (pvObj) Mutex();
00080      }
00081      return 0;
00082  }
00083  #endif
00084
00085  #if KERNEL_USE_EVENTFLAG
00086  //---------------------------------------------------------------------------
00087  EventFlag* AutoAlloc::NewEventFlag(void)
00088  {
00089      void* pvObj = Allocate(sizeof(EventFlag));
00090      if (pvObj) {
00091          return new (pvObj) EventFlag();
00092      }
00093      return 0;
00094  }
00095  #endif
00096
00097  #if KERNEL_USE_MESSAGE
00098  //---------------------------------------------------------------------------
00099  Message* AutoAlloc::NewMessage(void)
00100  {
00101      void* pvObj = Allocate(sizeof(Message));
00102      if (pvObj) {
00103          return new (pvObj) Message();
00104      }
00105      return 0;
00106  }
00107  //---------------------------------------------------------------------------
00108  MessageQueue* AutoAlloc::NewMessageQueue(void)
00109  {
00110      void* pvObj = Allocate(sizeof(MessageQueue));
00111      if (pvObj) {
00112          return new (pvObj) MessageQueue();
00113      }
00114      return 0;
00115  }
00116
00117  #endif
00118
00119  #if KERNEL_USE_NOTIFY
00120  //---------------------------------------------------------------------------
```

```
00121 Notify* AutoAlloc::NewNotify(void)
00122 {
00123     void* pvObj = Allocate(sizeof(Notify));
00124     if (pvObj) {
00125         return new (pvObj) Notify();
00126     }
00127     return 0;
00128 }
00129 #endif
00130
00131 #if KERNEL_USE_MAILBOX
00132 //---------------------------------------------------------------------------
00133 Mailbox* AutoAlloc::NewMailbox(void)
00134 {
00135     void* pvObj = Allocate(sizeof(Mailbox));
00136     if (pvObj) {
00137         return new (pvObj) Mailbox();
00138     }
00139     return 0;
00140 }
00141 #endif
00142
00143 //---------------------------------------------------------------------------
00144 Thread* AutoAlloc::NewThread(void)
00145 {
00146     void* pvObj = Allocate(sizeof(Thread));
00147     if (pvObj) {
00148         return new (pvObj) Thread();
00149     }
00150     return 0;
00151 }
00152
00153 #if KERNEL_USE_TIMERS
00154 //---------------------------------------------------------------------------
00155 Timer* AutoAlloc::NewTimer(void)
00156 {
00157     void* pvObj = Allocate(sizeof(Timer));
00158     if (pvObj) {
00159         return new (pvObj) Timer();
00160     }
00161     return 0;
00162 }
00163 #endif
00164
00165 #endif
```

20.5 /home/moslevin/mark3-source/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"
```

20.5.1 Detailed Description

Implementation of base class for blocking objects.

Definition in file blocking.cpp.

20.6 blocking.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_ \    |__ __|__|  __|_ __|  _____
00004 |    \  /  | ||    \      ||      |      ||  |/ /      ||___    |
```

```
00005 |     \/    | ||     \     ||    \     ||    \     ||___   |
00006 |__/\__/ |__|__||__|\__\   __||__|\__\   __||__|\__\   __||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------
00010
00011 Copyright (c) 2012-2016 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::BlockPriority(Thread* pclThread_)
00058 {
00059     KERNEL_ASSERT(pclThread_);
00060     KERNEL_TRACE_1("Blocking Thread %d", (uint16_t)pclThread_->
    GetID());
00061
00062     // Remove the thread from its current thread list (the "owner" list)
00063     // ... And add the thread to this object's block list
00064     Scheduler::Remove(pclThread_);
00065     m_clBlockList.AddPriority(pclThread_);
00066
00067     // Set the "current" list location to the blocklist for this thread
00068     pclThread_->SetCurrent(&m_clBlockList);
00069     pclThread_->SetState(THREAD_STATE_BLOCKED);
00070 }
00071
00072 //--------------------------------------------------------------------------
00073 void BlockingObject::UnBlock(Thread* pclThread_)
00074 {
00075     KERNEL_ASSERT(pclThread_);
00076     KERNEL_TRACE_1("Unblocking Thread %d", (uint16_t)pclThread_->
    GetID());
00077
00078     // Remove the thread from its current thread list (the "owner" list)
00079     pclThread_->GetCurrent()->Remove(pclThread_);
00080
00081     // Put the thread back in its active owner's list.  This is usually
00082     // the ready-queue at the thread's original priority.
00083     Scheduler::Add(pclThread_);
00084
00085     // Tag the thread's current list location to its owner
00086     pclThread_->SetCurrent(pclThread_->GetOwner());
00087     pclThread_->SetState(THREAD_STATE_READY);
00088 }
00089
00090 #endif
```

20.7 /home/moslevin/mark3-source/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>
```

20.7.1 Detailed Description

ATMega328p Profiling timer implementation.

Definition in file kernelprofile.cpp.

20.8 kernelprofile.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  |__    __|__  |__    __|__  |__    _____
00004 |    \  /   |  | |    \       | |    |       | |    | / /       | |___    |
00005 |     \/    |  | |    \       | |    |  \     | |    |  K   \      | |__     |
00006 |__/\__/|__|_||__|\__\   __|| |__|\__\   __|| |__|\__\   __|| |_____|
00007    |_____|      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 PORT_TIMER_COUNT_TYPE 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
```

20.9 /home/moslevin/mark3-source/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>
```

20.9.1 Detailed Description

Kernel Software interrupt implementation for ATMega328p.

Definition in file kernelswi.cpp.

20.10 kernelswi.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__    _|__  __|_    |_____
00004 |    \  /  |  ||    \     ||      |    ||  |/ /      ||___    |
00005 |     \/   |  ||     \     ||      ||      ||  |/      ||___    |
00006 |__/\__/|__|_||__|\__\   _||__|\__\   _||__|\__\   _||_____|
00007     |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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         EIMSK |= (1 << INT0);
00062     } else {
00063         EIMSK &= ~(1 << INT0);
00064     }
00065 }
00066
00067 //---------------------------------------------------------------------------
00068 void KernelSWI::Clear(void)
00069 {
00070     EIFR &= ~(1 << INTF0); // Clear the interrupt flag for INT0
00071 }
00072
00073 //---------------------------------------------------------------------------
00074 void KernelSWI::Trigger(void)
00075 {
00076     // if(Thread_IsSchedulerEnabled())
00077     {
00078         PORTD &= ~0x04;
00079         PORTD |= 0x04;
00080     }
00081 }
```

20.11 /home/moslevin/mark3-source/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>
```

20.11.1   Detailed Description

Kernel Timer Implementation for ATMega328p.

Definition in file kerneltimer.cpp.

20.12   kerneltimer.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
```

```
00003 ___|   _|__ __|_    |__ __|__   |__ __|__   |__ _____
00004 |   \ / | ||    \     ||    \     ||  |/ /     ||___  |
00005 |    \/  | ||    \    ||     \    ||   \       ||___   |
00006 |__/\__/|__|__||__|\__\ _||__|\__\ _||__|\__\ _||_____|
00007     |_____|     |_____|     |_____|     |_____|
00008
00009 --[Mark3 Realtime Platform]-----------------------------------------------
00010
00011 Copyright (c) 2012-2016 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  = ((PORT_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 PORT_TIMER_COUNT_TYPE 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 PORT_TIMER_COUNT_TYPE KernelTimer::SubtractExpiry(
00084 PORT_TIMER_COUNT_TYPE uInterval)
00085 {
00086 #if KERNEL_TIMERS_TICKLESS
00087     OCR1A -= uInterval;
00088     return OCR1A;
00089 #else
00090     return 0;
00091 #endif
00092 }
00093
00094 //--------------------------------------------------------------------------
00095 PORT_TIMER_COUNT_TYPE 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         return 0;
00103     } else {
00104         return (u16OCR1A - u16Read);
00105     }
00106 #else
00107     return 0;
00108 #endif
00109 }
00110
00111 //---------------------------------------------------------------------------
00112 PORT_TIMER_COUNT_TYPE KernelTimer::GetOvertime(void)
00113 {
00114     return KernelTimer::Read();
00115 }
00116
00117 //---------------------------------------------------------------------------
00118 PORT_TIMER_COUNT_TYPE KernelTimer::SetExpiry(uint32_t
    u32Interval_)
00119 {
00120 #if KERNEL_TIMERS_TICKLESS
00121     uint16_t u16SetInterval;
00122     if (u32Interval_ > 65535) {
00123         u16SetInterval = 65535;
00124     } else {
00125         u16SetInterval = (uint16_t)u32Interval_;
00126     }
00127
00128     OCR1A = u16SetInterval;
00129     return u16SetInterval;
00130 #else
00131     return 0;
00132 #endif
00133 }
00134
00135 //---------------------------------------------------------------------------
00136 void KernelTimer::ClearExpiry(void)
00137 {
00138 #if KERNEL_TIMERS_TICKLESS
00139     OCR1A = 65535; // Clear the compare value
00140 #endif
00141 }
00142
00143 //---------------------------------------------------------------------------
00144 uint8_t KernelTimer::DI(void)
00145 {
00146 #if KERNEL_TIMERS_TICKLESS
00147     bool bEnabled = ((TIMSK1 & (TIMER_IMSK)) != 0);
00148     TIFR1 &= ~TIMER_IFR;   // Clear interrupt flags
00149     TIMSK1 &= ~TIMER_IMSK; // Disable interrupt
00150     return bEnabled;
00151 #else
00152     return 0;
00153 #endif
00154 }
00155
00156 //---------------------------------------------------------------------------
00157 void KernelTimer::EI(void)
00158 {
00159     KernelTimer::RI(0);
00160 }
00161
00162 //---------------------------------------------------------------------------
00163 void KernelTimer::RI(bool bEnable_)
00164 {
00165 #if KERNEL_TIMERS_TICKLESS
00166     if (bEnable_) {
00167         TIMSK1 |= (1 << OCIE1A); // Enable interrupt
00168     } else {
00169         TIMSK1 &= ~(1 << OCIE1A);
00170     }
00171 #endif
00172 }
```

20.13 /home/moslevin/mark3-source/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.

20.13.1 Detailed Description

Profiling timer hardware interface.

Definition in file kernelprofile.h.

20.14 kernelprofile.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /    | ||    \   ||    |    |    ||    |/ /    ||___  |
00005 |     \/     | ||     \   ||    |    \   ||    |    \    ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 PORT_TIMER_COUNT_TYPE Read();
00068
00074     static void Process();
00075
00081     static uint32_t GetEpoch() { return m_u32Epoch; }
00082 private:
00083     static uint32_t m_u32Epoch;
00084 };
00085
00086 #endif // KERNEL_USE_PROFILER
00087
00088 #endif
```

20.15 /home/moslevin/mark3-source/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.

20.15.1 Detailed Description

Kernel Software interrupt declarations.

Definition in file kernelswi.h.

20.16 kernelswi.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__   __|_   |__    __|_   |__    __|_   |__    __|_____
00004 |    \  /   |  ||    \      ||    |      ||   |/ /      ||___    |
00005 |     \/    |  ||     \     ||    |      ||   |  \      ||___    |
00006 |__/\__/|__|_||__|\__\   __||__|\__\   __||__|\__\   __||_____|
00007     |____|      |____|       |____|       |____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #include "kerneltypes.h"
00023 #ifndef __KERNELSWI_H_
00024 #define __KERNELSWI_H_
00025
00026 //---------------------------------------------------------------------------
00031 class KernelSWI
00032 {
00033 public:
00040     static void Config(void);
00041
00047     static void Start(void);
00048
00054     static void Stop(void);
00055
00061     static void Clear(void);
00062
00069     static void Trigger(void);
00070
00078     static uint8_t DI();
00079
00087     static void RI(bool bEnable_);
00088 };
00089
00090 #endif // __KERNELSIW_H_
```

20.17 /home/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/kerneltimer.h File Reference

Kernel Timer Class declaration.

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

Classes

- class KernelTimer

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

20.17.1 Detailed Description

Kernel Timer Class declaration.

Definition in file kerneltimer.h.

20.18 kerneltimer.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____        _____
00003  ___|    _|__  __|_    |_    |__   __|_    |__   __|_    |__   _____
00004 |    \  /   | ||    \       ||       |     ||  |/ /        ||___   |
00005 |     \/    | ||     \      ||       \     ||  |  \        ||___   |
00006 |__/\__/|__|_||__|\__\   __||__|\__\   __||__|\__\   __||_____|
00007     |____|       |____|         |____|        |____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========================================================================== */
00021 #include "kerneltypes.h"
00022 #include "mark3cfg.h"
00023
00024 #ifndef __KERNELTIMER_H_
00025 #define __KERNELTIMER_H_
00026
00027 //---------------------------------------------------------------------------
00031 class KernelTimer
00032 {
00033 public:
00039     static void Config(void);
00040
00046     static void Start(void);
00047
00053     static void Stop(void);
00054
00060     static uint8_t DI(void);
00061
00069     static void RI(bool bEnable_);
00070
00076     static void EI(void);
00077
00088     static PORT_TIMER_COUNT_TYPE SubtractExpiry(
00     PORT_TIMER_COUNT_TYPE uInterval_);
00089
00098     static PORT_TIMER_COUNT_TYPE TimeToExpiry(void);
00099
00108     static PORT_TIMER_COUNT_TYPE SetExpiry(uint32_t u32Interval_);
00109
00118     static PORT_TIMER_COUNT_TYPE GetOvertime(void);
00119
00125     static void ClearExpiry(void);
00126
00134     static PORT_TIMER_COUNT_TYPE Read(void);
00135 };
00136
00137 #endif //__KERNELTIMER_H_
```

20.19 /home/moslevin/mark3-source/embedded/kernel/cpu/avr/atmega328p/gcc/public/portcfg.h File Reference

Mark3 Port Configuration.

Macros

- #define AVR (1)

  Define a macro indicating the CPU architecture for which this port belongs.
- #define K_WORD uint8_t

  Define types that map to the CPU Architecture's default data-word and address size.
- #define K_ADDR uint16_t

  Size of an address (pointer size)
- #define PORT_PRIO_TYPE uint8_t

  Set a base datatype used to represent each element of the scheduler's priority bitmap.
- #define PORT_PRIO_MAP_WORD_SIZE (1)

  size of PORT_PRIO_TYPE in bytes
- #define PORT_SYSTEM_FREQ ((uint32_t)16000000)

  Define the running CPU frequency.
- #define PORT_TIMER_FREQ ((uint32_t)(PORT_SYSTEM_FREQ / 1000))

  Set the timer frequency.
- #define PORT_TIMER_COUNT_TYPE uint16_t

  Define the native type corresponding to the kernel timer hardware's counter register.
- #define PORT_MIN_TIMER_TICKS (0)

  Minimum number of timer ticks for any delay or sleep, required to ensure that a timer cannot be initialized to a negative value.

20.19.1 Detailed Description

Mark3 Port Configuration.

This file is used to configure the kernel for your specific target CPU in order to provide the optimal set of features for a given use case.

!! NOTE: This file must ONLY be included from mark3cfg.h

Definition in file portcfg.h.

20.19.2 Macro Definition Documentation

20.19.2.1 #define AVR (1)

Define a macro indicating the CPU architecture for which this port belongs.

This may also be set by the toolchain, but that's not guaranteed.

Definition at line 34 of file portcfg.h.

20.19.2.2 #define K_WORD uint8_t

Define types that map to the CPU Architecture's default data-word and address size.

Size of a data word

Examples:

buffalogger/main.cpp, lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_idle_function/main.cpp, lab3_round_robin/main.cpp, lab4_semaphores/main.cpp, lab5_mutexes/main.↩ cpp, lab6_timers/main.cpp, lab7_events/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_↩ threads/main.cpp.

Definition at line 41 of file portcfg.h.

20.19.2.3 #define PORT_PRIO_TYPE uint8_t

Set a base datatype used to represent each element of the scheduler's priority bitmap.

PORT_PRIO_MAP_WORD_SIZE should map to the *size* of an element of type PORT_PROI_TYPE.Type used for bitmap in the PriorityMap class

Definition at line 51 of file portcfg.h.

20.19.2.4 #define PORT_SYSTEM_FREQ ((uint32_t)16000000)

Define the running CPU frequency.

This may be an integer constant, or an alias for another variable which holds the CPU's current running frequency.↩ CPU Frequency in Hz

Definition at line 59 of file portcfg.h.

20.19.2.5 #define PORT_TIMER_COUNT_TYPE uint16_t

Define the native type corresponding to the kernel timer hardware's counter register.

Timer counter type

Definition at line 78 of file portcfg.h.

20.19.2.6 #define PORT_TIMER_FREQ ((uint32_t)(**PORT_SYSTEM_FREQ** / 1000))

Set the timer frequency.

If running in tickless mode, this is simply the frequency at which the free-running kernel timer increments.

In tick-based mode, this is the frequency at which the fixed-frequency kernel tick interrupt occurs.Fixed timer interrupt frequency

Definition at line 72 of file portcfg.h.

20.20 portcfg.h

```
00001  /*===========================================================================
00002        _____        _____        _____        _____
00003   ___|   _|__  __|_   |__  __|__   |__  __|__   |__  _____
00004  |     \ /   |  |  |      \       |  |    |/ /      ||___   |
00005  |      \/   | ||       \      ||       \      ||    \      ||___    |
00006  |__/\__/|__|__||__|\__\   _||__|\__\   _||__|\__\   _||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009  --[Mark3 Realtime Platform]-------------------------------------------
00010
00011  Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012  See license.txt for more information
00013  ===========================================================================*/
00025  #ifndef __PORTCFG_H__
00026  #define __PORTCFG_H__
00027
```

```
00033 #ifndef AVR
00034 # define AVR                         (1)
00035 #endif
00036
00041 #define K_WORD                       uint8_t
00042 #define K_ADDR                       uint16_t
00043
00044
00051 #define PORT_PRIO_TYPE               uint8_t
00052 #define PORT_PRIO_MAP_WORD_SIZE      (1)
00053
00054
00058 #if !defined(PORT_SYSTEM_FREQ)
00059 #define PORT_SYSTEM_FREQ             ((uint32_t)16000000)
00060 #endif
00061
00069 #if KERNEL_TIMERS_TICKLESS
00070 #define PORT_TIMER_FREQ              ((uint32_t)(PORT_SYSTEM_FREQ / 256))
00071 #else
00072 #define PORT_TIMER_FREQ              ((uint32_t)(PORT_SYSTEM_FREQ / 1000))
00073 #endif
00074
00078 #define PORT_TIMER_COUNT_TYPE        uint16_t
00079
00080
00084 #define PORT_MIN_TIMER_TICKS         (0)
00085
00086 #endif // __PORTCFG_H__
```

20.21 /home/moslevin/mark3-source/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)

  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.

Functions

- uint8_t __mark3_clz8 (uint8_t in_)

20.21.1   Detailed Description

ATMega328p Multithreading support.

Definition in file threadport.h.

20.21.2   Macro Definition Documentation

20.21.2.1   #define CS_ENTER(   )

Value:

```
\
{                                   \
                        \
uint8_t __x = _SFR_IO8(SR_);        \
                                 \
ASM("cli");
```

These macros *must* be used in pairs !

Enter critical section (copy status register, disable interrupts)

Examples:

buffalogger/main.cpp, and lab9_dynamic_threads/main.cpp.

Definition at line 245 of file threadport.h.

20.21.3   Function Documentation
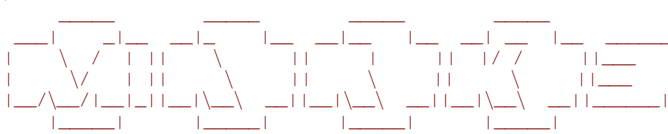
20.21.3.1   uint8_t __mark3_clz8 (uint8_t *in_*)   [inline]

Lookup table based count-leading zeros implementation, used by scheduler by way of PriorityMap

Definition at line 51 of file threadport.h.

20.22   threadport.h

```
00001 /*========================================================================
00002        _____        _____        _____        _____
00003   ___|    _|__    __|_    |__    |__ _|__   __|_    |__   |__ _____
00004  |    \  /  |  |  |    \     |    |  |  |/ /     |    |___  |
00005  |     \/   |  | |        \     |    |  |     \      |    |___   |
00006  |__/\__/|__|_|_|__|\__\  __|__|\__\  __|__|_\__\  __|__|_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
```

```
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 //-------------------------------------------------------------------------------
00041 #define TOP_OF_STACK(x, y) (uint8_t*)(((uint16_t)x) + (y - 1))
00042 #define PUSH_TO_STACK(x, y)                                                     \
00044     *x = y;                                                                     \
00045     x--;
00046 #define STACK_GROWS_DOWN (1)
00047
00048 //-------------------------------------------------------------------------------
00051 inline uint8_t __mark3_clz8(uint8_t in_)
00052 {
00053     static const uint8_t u8Lookup[] = {4, 3, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0};
00054     uint8_t hi = __builtin_avr_swap(in_) & 0x0F;
00055     if (hi) {
00056         return u8Lookup[hi];
00057     }
00058     return 4 + u8Lookup[in_];
00059 }
00060
00061 //-------------------------------------------------------------------------------
00062 #define HW_CLZ   (1)
00063 #define CLZ(x)   __mark3_clz8(x)
00064
00065 //-------------------------------------------------------------------------------
00067 #define Thread_SaveContext()                                                    \
00068                                                                                 \
00069 ASM("push r0");                                                                 \
00070                                                                                 \
00071 ASM("in r0, __SREG__");                                                         \
00072                                                                                 \
00073 ASM("cli");                                                                     \
00074                                                                                 \
00075 ASM("push r0");                                                                 \
00076                                                                                 \
00077 ASM("push r1");                                                                 \
00078                                                                                 \
00079 ASM("clr r1");                                                                  \
00080                                                                                 \
00081 ASM("push r2");                                                                 \
00082                                                                                 \
00083 ASM("push r3");                                                                 \
00084                                                                                 \
00085 ASM("push r4");                                                                 \
00086                                                                                 \
00087 ASM("push r5");                                                                 \
00088                                                                                 \
00089 ASM("push r6");                                                                 \
00090                                                                                 \
00091 ASM("push r7");                                                                 \
00092                                                                                 \
00093 ASM("push r8");                                                                 \
00094                                                                                 \
```

```
00095 ASM("push r9");
                        \
00096       \
00097 ASM("push r10");
                        \
00098       \
00099 ASM("push r11");
                        \
00100       \
00101 ASM("push r12");
                        \
00102       \
00103 ASM("push r13");
                        \
00104       \
00105 ASM("push r14");
                        \
00106       \
00107 ASM("push r15");
                        \
00108       \
00109 ASM("push r16");
                        \
00110       \
00111 ASM("push r17");
                        \
00112       \
00113 ASM("push r18");
                        \
00114       \
00115 ASM("push r19");
                        \
00116       \
00117 ASM("push r20");
                        \
00118       \
00119 ASM("push r21");
                        \
00120       \
00121 ASM("push r22");
                        \
00122       \
00123 ASM("push r23");
                        \
00124       \
00125 ASM("push r24");
                        \
00126       \
00127 ASM("push r25");
                        \
00128       \
00129 ASM("push r26");
                        \
00130       \
00131 ASM("push r27");
                        \
00132       \
00133 ASM("push r28");
                        \
00134       \
00135 ASM("push r29");
                        \
00136       \
00137 ASM("push r30");
                        \
00138       \
00139 ASM("push r31");
                        \
00140       \
00141 ASM("lds r26, g_pclCurrent");
                        \
00142       \
00143 ASM("lds r27, g_pclCurrent + 1");
                        \
00144       \
00145 ASM("adiw r26, 4");
                        \
00146       \
00147 ASM("in    r0, 0x3D");
                        \
00148       \
00149 ASM("st    x+, r0");
                        \
00150       \
00151 ASM("in    r0, 0x3E");
                        \
00152       \
```

```
00153 ASM("st    x+, r0");
00154
00155 //---------------------------------------------------------------------------
00157 #define Thread_RestoreContext()                                      \
00158                                                                      \
00159 ASM("lds r26, g_pclCurrent");                                        \
00160                                                                      \
00161 ASM("lds r27, g_pclCurrent + 1");                                    \
00162                                                                      \
00163 ASM("adiw r26, 4");                                                  \
00164                                                                      \
00165 ASM("ld    r28, x+");                                                \
00166                                                                      \
00167 ASM("out 0x3D, r28");                                                \
00168                                                                      \
00169 ASM("ld    r29, x+");                                                \
00170                                                                      \
00171 ASM("out 0x3E, r29");                                                \
00172                                                                      \
00173 ASM("pop r31");                                                      \
00174                                                                      \
00175 ASM("pop r30");                                                      \
00176                                                                      \
00177 ASM("pop r29");                                                      \
00178                                                                      \
00179 ASM("pop r28");                                                      \
00180                                                                      \
00181 ASM("pop r27");                                                      \
00182                                                                      \
00183 ASM("pop r26");                                                      \
00184                                                                      \
00185 ASM("pop r25");                                                      \
00186                                                                      \
00187 ASM("pop r24");                                                      \
00188                                                                      \
00189 ASM("pop r23");                                                      \
00190                                                                      \
00191 ASM("pop r22");                                                      \
00192                                                                      \
00193 ASM("pop r21");                                                      \
00194                                                                      \
00195 ASM("pop r20");                                                      \
00196                                                                      \
00197 ASM("pop r19");                                                      \
00198                                                                      \
00199 ASM("pop r18");                                                      \
00200                                                                      \
00201 ASM("pop r17");                                                      \
00202                                                                      \
00203 ASM("pop r16");                                                      \
00204                                                                      \
00205 ASM("pop r15");                                                      \
00206                                                                      \
00207 ASM("pop r14");                                                      \
00208                                                                      \
00209 ASM("pop r13");                                                      \
00210                                                                      \
00211 ASM("pop r12");                                                      \
00212                                                                      \
```

```
00213 ASM("pop r11");
                    \
00214       \
00215 ASM("pop r10");
                    \
00216       \
00217 ASM("pop r9");
                    \
00218       \
00219 ASM("pop r8");
                    \
00220       \
00221 ASM("pop r7");
                    \
00222       \
00223 ASM("pop r6");
                    \
00224       \
00225 ASM("pop r5");
                    \
00226       \
00227 ASM("pop r4");
                    \
00228       \
00229 ASM("pop r3");
                    \
00230       \
00231 ASM("pop r2");
                    \
00232       \
00233 ASM("pop r1");
                    \
00234       \
00235 ASM("pop r0");
                    \
00236       \
00237 ASM("out __SREG__, r0");
                    \
00238       \
00239 ASM("pop r0");
00240
00241 //-------------------------------------------------------------------------
00243 //-------------------------------------------------------------------------
00245 #define CS_ENTER()
                    \
00246       \
00247 {
                    \
00248       \
00249 uint8_t __x = _SFR_IO8(SR_);
                    \
00250       \
00251 ASM("cli");
00252 //-------------------------------------------------------------------------
00254 #define CS_EXIT()
                    \
00255       \
00256 _SFR_IO8(SR_)
                    \
00257       = __x;
                    \
00258       \
00259 }
00260
00261 //-------------------------------------------------------------------------
00263 #define ENABLE_INTS() ASM("sei");
00264 #define DISABLE_INTS() ASM("cli");
00265
00266 //-------------------------------------------------------------------------
00267 class Thread;
00275 class ThreadPort
00276 {
00277 public:
00283     static void StartThreads();
00284     friend class Thread;
00285
00286 private:
00294     static void InitStack(Thread* pstThread_);
00295 };
00296
00297 #endif //__ThreadPORT_H_
```

20.23 /home/moslevin/mark3-source/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

  ISR(INT0_vect) SWI using INT0 - used to trigger a context switch.
- ISR (TIMER1_COMPA_vect)

  ISR(TIMER1_COMPA_vect) Timer interrupt ISR - causes a tick, which may cause a context switch.

20.23.1 Detailed Description

ATMega328p Multithreading.

Definition in file threadport.cpp.

20.24 threadport.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__    |     |__    |     |__    |     |__    |     |_____
00004 |    \  /   |  | |   \   |   ||    |   ||    ||   |/ /      ||___     |
00005 |     \/    |  | |    \    |   ||    |   ||    ||   |\ \      ||___     |
00006 |__/\__/|__|__||__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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         pclThread_->m_pwStack[i] = 0xFF;
00056     }
00057
00058     // Our context starts with the entry function
00059     PUSH_TO_STACK(pu8Stack, (uint8_t)(u16Addr & 0x00FF));
00060     PUSH_TO_STACK(pu8Stack, (uint8_t)((u16Addr >> 8) & 0x00FF));
00061
00062     // R0
00063     PUSH_TO_STACK(pu8Stack, 0x00); // R0
00064
00065     // Push status register and R1 (which is used as a constant zero)
00066     PUSH_TO_STACK(pu8Stack, 0x80); // SR
00067     PUSH_TO_STACK(pu8Stack, 0x00); // R1
00068
00069     // Push other registers
00070     for (i = 2; i <= 23; i++) // R2-R23
00071     {
00072         PUSH_TO_STACK(pu8Stack, i);
00073     }
00074
00075     // Assume that the argument is the only stack variable
00076     PUSH_TO_STACK(pu8Stack, (uint8_t)(((uint16_t)(pclThread_->
    m_pvArg)) & 0x00FF));          // R24
00077     PUSH_TO_STACK(pu8Stack, (uint8_t)((((uint16_t)(pclThread_->
    m_pvArg)) >> 8) & 0x00FF)); // R25
00078
00079     // Push the rest of the registers in the context
00080     for (i = 26; i <= 31; i++) {
00081         PUSH_TO_STACK(pu8Stack, i);
00082     }
00083
00084     // Set the top o' the stack.
00085     pclThread_->m_pwStackTop = (uint8_t*)pu8Stack;
00086
00087     // That's it!  the thread is ready to run now.
00088 }
00089
00090 //---------------------------------------------------------------------------
00091 static void Thread_Switch(void)
00092 {
00093 #if KERNEL_USE_IDLE_FUNC
00094     // If there's no next-thread-to-run...
00095     if (g_pclNext == Kernel::GetIdleThread()) {
00096         g_pclCurrent = Kernel::GetIdleThread();
00097
00098         // Disable the SWI, and re-enable interrupts -- enter nested interrupt
00099         // mode.
00100         KernelSWI::DI();
00101
00102         uint8_t u8SR = _SFR_IO8(SR_);
00103
00104         // So long as there's no "next-to-run" thread, keep executing the Idle
00105         // function to conclusion...
00106
00107         while (g_pclNext == Kernel::GetIdleThread()) {
00108             // Ensure that we run this block in an interrupt enabled context (but
00109             // with the rest of the checks being performed in an interrupt disabled
00110             // context).
00111             ASM("sei");
00112             Kernel::IdleFunc();
00113             ASM("cli");
00114         }
00115
00116         // Progress has been achieved -- an interrupt-triggered event has caused
00117         // the scheduler to run, and choose a new thread.  Since we've already
00118         // saved the context of the thread we've hijacked to run idle, we can
00119         // proceed to disable the nested interrupt context and switch to the
00120         // new thread.
00121
00122         _SFR_IO8(SR_) = u8SR;
00123         KernelSWI::RI(true);
00124     }
```

```
00125 #endif
00126     g_pclCurrent = (Thread*)g_pclNext;
00127 }
00128
00129 //---------------------------------------------------------------------------
00130 void ThreadPort::StartThreads()
00131 {
00132     KernelSWI::Config();   // configure the task switch SWI
00133     KernelTimer::Config(); // configure the kernel timer
00134
00135     Scheduler::SetScheduler(1); // enable the scheduler
00136     Scheduler::Schedule();      // run the scheduler - determine the first thread to run
00137
00138     Thread_Switch(); // Set the next scheduled thread to the current thread
00139
00140     KernelTimer::Start(); // enable the kernel timer
00141     KernelSWI::Start();   // enable the task switch SWI
00142
00143 #if KERNEL_USE_QUANTUM
00144     // Restart the thread quantum timer, as any value held prior to starting
00145     // the kernel will be invalid.  This fixes a bug where multiple threads
00146     // started with the highest priority before starting the kernel causes problems
00147     // until the running thread voluntarily blocks.
00148     Quantum::RemoveThread();
00149     Quantum::AddThread(g_pclCurrent);
00150 #endif
00151
00152     // Restore the context...
00153     Thread_RestoreContext(); // restore the context of the first running thread
00154     ASM("reti");             // return from interrupt - will return to the first scheduled thread
00155 }
00156
00157 //---------------------------------------------------------------------------
00162 //---------------------------------------------------------------------------
00163 ISR(INT0_vect) __attribute__((signal, naked));
00164 ISR(INT0_vect)
00165 {
00166     Thread_SaveContext();    // Push the context (registers) of the current task
00167     Thread_Switch();         // Switch to the next task
00168     Thread_RestoreContext(); // Pop the context (registers) of the next task
00169     ASM("reti");             // Return to the next task
00170 }
00171
00172 //---------------------------------------------------------------------------
00177 //---------------------------------------------------------------------------
00178 ISR(TIMER1_COMPA_vect)
00179 {
00180 #if KERNEL_USE_TIMERS
00181     TimerScheduler::Process();
00182 #endif
00183 #if KERNEL_USE_QUANTUM
00184     Quantum::UpdateTimer();
00185 #endif
00186 }
```

20.25 /home/moslevin/mark3-source/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".

20.25.1 Detailed Description

Device driver/hardware abstraction layer.

Definition in file driver.cpp.

20.25.2 Function Documentation

20.25.2.1 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 75 of file driver.cpp.

20.26 driver.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /  |  | ||    \       ||    |       ||   |/ /       ||___  |
00005 |     \/   |  | ||     \      ||    |\      ||   |  \       ||__   |
00006 |__/\__/|__|__||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007      |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]---------------------------------------------
00010
00011 Copyright (c) 2012-2016 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     virtual uint16_t Read(uint16_t u16Bytes_, uint8_t* pu8Data_) { return 0; }
00053     virtual uint16_t Write(uint16_t u16Bytes_, uint8_t* pu8Data_) { return 0; }
00054     virtual uint16_t
00055     Control(uint16_t u16Event_, void* pvDataIn_, uint16_t u16SizeIn_, void* pvDataOut_, uint16_t
    u16SizeOut_)
00056     {
00057         return 0;
00058     }
00059 };
00060
00061 //---------------------------------------------------------------------------
00062 static DevNull clDevNull;
00063
00064 //---------------------------------------------------------------------------
00075 static uint8_t DrvCmp(const char* szStr1_, const char* szStr2_)
00076 {
00077     char* szTmp1 = (char*)szStr1_;
00078     char* szTmp2 = (char*)szStr2_;
00079
00080     while (*szTmp1 && *szTmp2) {
00081         if (*szTmp1++ != *szTmp2++) {
00082             return 0;
00083         }
00084     }
00085
00086     // Both terminate at the same length
00087     if (!(*szTmp1) && !(*szTmp2)) {
00088         return 1;
00089     }
00090
00091     return 0;
00092 }
00093
00094 //---------------------------------------------------------------------------
00095 void DriverList::Init()
00096 {
00097     // Ensure we always have at least one entry - a default in case no match
00098     // is found (/dev/null)
00099     clDevNull.Init();
00100     Add(&clDevNull);
00101 }
00102
00103 //---------------------------------------------------------------------------
00104 Driver* DriverList::FindByPath(const char* m_pcPath)
00105 {
00106     KERNEL_ASSERT(m_pcPath);
00107     Driver* pclTemp = static_cast<Driver*>(m_clDriverList.
    GetHead());
00108
00109     // Iterate through the list of drivers until we find a match, or we
00110     // exhaust our list of installed drivers
00111     while (pclTemp) {
00112         if (DrvCmp(m_pcPath, pclTemp->GetPath())) {
00113             return pclTemp;
00114         }
00115         pclTemp = static_cast<Driver*>(pclTemp->GetNext());
00116     }
00117     // No matching driver found - return a pointer to our /dev/null driver
00118     return &clDevNull;
00119 }
00120
00121 #endif
```

20.27 /home/moslevin/mark3-source/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.

20.27.1   Detailed Description

Event Flag Blocking Object/IPC-Object implementation.

Definition in file eventflag.cpp.

20.27.2   Function Documentation

20.27.2.1   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.

20.28   eventflag.cpp

```
00001 /*=========================================================================
00002        _____        _____        _____        _____
00003  ___|    _|__    __|_    |__    |__    _|__    |__    _____
00004  |      \  /   |  ||       \        ||        |       ||  |/  /       ||___    |
00005  |        \/    |  ||        \        ||        __\        ||        \        ||___     |
00006  |__/\__/|__|__||__|\__\    _||__|\__\    _||__|\__\    _||_____|
00007      |____|        |____|        |____|        |____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------
00010
00011 Copyright (c) 2012-2016 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
00062     ()->GetCurPriority()) {
00062         Thread::Yield();
00063     }
00064 }
00065 //---------------------------------------------------------------------------
00066 EventFlag::~EventFlag()
00067 {
00068     // If there are any threads waiting on this object when it goes out
00069     // of scope, set a kernel panic.
00070     if (m_clBlockList.HighestWaiter()) {
00071         Kernel::Panic(PANIC_ACTIVE_EVENTFLAG_DESCOPED);
00072     }
00073 }
00074
00075 //---------------------------------------------------------------------------
00076 void EventFlag::WakeMe(Thread* pclChosenOne_)
00077 {
00078     UnBlock(pclChosenOne_);
00079 }
00080 #endif
00081
00082 //---------------------------------------------------------------------------
00083 #if KERNEL_USE_TIMEOUTS
00084 uint16_t EventFlag::Wait_i(uint16_t u16Mask_,
00084 EventFlagOperation_t eMode_, uint32_t u32TimeMS_)
00085 #else
00086 uint16_t EventFlag::Wait_i(uint16_t u16Mask_,
00086 EventFlagOperation_t eMode_)
00087 #endif
00088 {
00089     bool bThreadYield = false;
00090     bool bMatch      = false;
00091
00092 #if KERNEL_USE_TIMEOUTS
00093     Timer clEventTimer;
00094     bool  bUseTimer = false;
00095 #endif
00096
00097     // Ensure we're operating in a critical section while we determine
00098     // whether or not we need to block the current thread on this object.
00099     CS_ENTER();
00100
00101     // Check to see whether or not the current mask matches any of the
00102     // desired bits.
00103     g_pclCurrent->SetEventFlagMask(u16Mask_);
00104
00105     if ((eMode_ == EVENT_FLAG_ALL) || (eMode_ ==
00105     EVENT_FLAG_ALL_CLEAR)) {
00106         // Check to see if the flags in their current state match all of
00107         // the set flags in the event flag group, with this mask.
00108         if ((m_u16SetMask & u16Mask_) == u16Mask_) {
00109             bMatch = true;
00110             g_pclCurrent->SetEventFlagMask(u16Mask_);
00111         }
00112     } else if ((eMode_ == EVENT_FLAG_ANY) || (eMode_ ==
00112     EVENT_FLAG_ANY_CLEAR)) {
00113         // Check to see if the existing flags match any of the set flags in
00114         // the event flag group  with this mask
00115         if (m_u16SetMask & u16Mask_) {
00116             bMatch = true;
00117             g_pclCurrent->SetEventFlagMask(m_u16SetMask & u16Mask_);
00118         }
00119     }
00120
```

```
00121      // We're unable to match this pattern as-is, so we must block.
00122      if (!bMatch) {
00123          // Reset the current thread's event flag mask & mode
00124          g_pclCurrent->SetEventFlagMask(u16Mask_);
00125          g_pclCurrent->SetEventFlagMode(eMode_);
00126
00127 #if KERNEL_USE_TIMEOUTS
00128          if (u32TimeMS_) {
00129              g_pclCurrent->SetExpired(false);
00130              clEventTimer.Init();
00131              clEventTimer.Start(0, u32TimeMS_, TimedEventFlag_Callback, (void*)this);
00132              bUseTimer = true;
00133          }
00134 #endif
00135
00136          // Add the thread to the object's block-list.
00137          BlockPriority(g_pclCurrent);
00138
00139          // Trigger that
00140          bThreadYield = true;
00141      }
00142
00143      // If bThreadYield is set, it means that we've blocked the current thread,
00144      // and must therefore rerun the scheduler to determine what thread to
00145      // switch to.
00146      if (bThreadYield) {
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          clEventTimer.Stop();
00161      }
00162 #endif
00163
00164      return g_pclCurrent->GetEventFlagMask();
00165 }
00166
00167 //---------------------------------------------------------------------------
00168 uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_)
00169 {
00170 #if KERNEL_USE_TIMEOUTS
00171      return Wait_i(u16Mask_, eMode_, 0);
00172 #else
00173      return Wait_i(u16Mask_, eMode_);
00174 #endif
00175 }
00176
00177 #if KERNEL_USE_TIMEOUTS
00178 //---------------------------------------------------------------------------
00179 uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_,
00     uint32_t u32TimeMS_)
00180 {
00181      return Wait_i(u16Mask_, eMode_, u32TimeMS_);
00182 }
00183 #endif
00184
00185 //---------------------------------------------------------------------------
00186 void EventFlag::Set(uint16_t u16Mask_)
00187 {
00188      Thread*  pclPrev;
00189      Thread*  pclCurrent;
00190      bool     bReschedule = false;
00191      uint16_t u16NewMask;
00192
00193      CS_ENTER();
00194
00195      // Walk through the whole block list, checking to see whether or not
00196      // the current flag set now matches any/all of the masks and modes of
00197      // the threads involved.
00198
00199      m_u16SetMask |= u16Mask_;
00200      u16NewMask = m_u16SetMask;
00201
00202      // Start at the head of the list, and iterate through until we hit the
00203      // "head" element in the list again.  Ensure that we handle the case where
00204      // we remove the first or last elements in the list, or if there's only
00205      // one element in the list.
00206      pclCurrent = static_cast<Thread*>(m_clBlockList.GetHead());
00207
00208      // Do nothing when there are no objects blocking.
00209      if (pclCurrent) {
00210          // First loop - process every thread in the block-list and check to
```

```
00211            // see whether or not the current flags match the event-flag conditions
00212            // on the thread.
00213            do {
00214                pclPrev    = pclCurrent;
00215                pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00216
00217                // Read the thread's event mask/mode
00218                uint16_t             u16ThreadMask = pclPrev->GetEventFlagMask();
00219                EventFlagOperation_t eThreadMode   = pclPrev->
     GetEventFlagMode();
00220
00221                // For the "any" mode - unblock the blocked threads if one or more bits
00222                // in the thread's bitmask match the object's bitmask
00223                if ((EVENT_FLAG_ANY == eThreadMode) || (
     EVENT_FLAG_ANY_CLEAR == eThreadMode)) {
00224                    if (u16ThreadMask & m_u16SetMask) {
00225                        pclPrev->SetEventFlagMode(
     EVENT_FLAG_PENDING_UNBLOCK);
00226                        pclPrev->SetEventFlagMask(m_u16SetMask & u16ThreadMask);
00227                        bReschedule = true;
00228
00229                        // If the "clear" variant is set, then clear the bits in the mask
00230                        // that caused the thread to unblock.
00231                        if (EVENT_FLAG_ANY_CLEAR == eThreadMode) {
00232                            u16NewMask &= ~(u16ThreadMask & u16Mask_);
00233                        }
00234                    }
00235                }
00236                // For the "all" mode, every set bit in the thread's requested bitmask must
00237                // match the object's flag mask.
00238                else if ((EVENT_FLAG_ALL == eThreadMode) || (
     EVENT_FLAG_ALL_CLEAR == eThreadMode)) {
00239                    if ((u16ThreadMask & m_u16SetMask) == u16ThreadMask) {
00240                        pclPrev->SetEventFlagMode(
     EVENT_FLAG_PENDING_UNBLOCK);
00241                        pclPrev->SetEventFlagMask(u16ThreadMask);
00242                        bReschedule = true;
00243
00244                        // If the "clear" variant is set, then clear the bits in the mask
00245                        // that caused the thread to unblock.
00246                        if (EVENT_FLAG_ALL_CLEAR == eThreadMode) {
00247                            u16NewMask &= ~(u16ThreadMask & u16Mask_);
00248                        }
00249                    }
00250                }
00251            }
00252            // To keep looping, ensure that there's something in the list, and
00253            // that the next item isn't the head of the list.
00254            while (pclPrev != m_clBlockList.GetTail());
00255
00256            // Second loop - go through and unblock all of the threads that
00257            // were tagged for unblocking.
00258            pclCurrent   = static_cast<Thread*>(m_clBlockList.
     GetHead());
00259            bool bIsTail = false;
00260            do {
00261                pclPrev    = pclCurrent;
00262                pclCurrent = static_cast<Thread*>(pclCurrent->GetNext());
00263
00264                // Check to see if this is the condition to terminate the loop
00265                if (pclPrev == m_clBlockList.GetTail()) {
00266                    bIsTail = true;
00267                }
00268
00269                // If the first pass indicated that this thread should be
00270                // unblocked, then unblock the thread
00271                if (pclPrev->GetEventFlagMode() ==
     EVENT_FLAG_PENDING_UNBLOCK) {
00272                    UnBlock(pclPrev);
00273                }
00274            } while (!bIsTail);
00275        }
00276
00277        // If we awoke any threads, re-run the scheduler
00278        if (bReschedule) {
00279            Thread::Yield();
00280        }
00281
00282        // Update the bitmask based on any "clear" operations performed along
00283        // the way
00284        m_u16SetMask = u16NewMask;
00285
00286        // Restore interrupts - will potentially cause a context switch if a
00287        // thread is unblocked.
00288        CS_EXIT();
00289 }
00290
```

```
00291 //---------------------------------------------------------------------------
00292 void EventFlag::Clear(uint16_t u16Mask_)
00293 {
00294     // Just clear the bitfields in the local object.
00295     CS_ENTER();
00296     m_u16SetMask &= ~u16Mask_;
00297     CS_EXIT();
00298 }
00299
00300 //---------------------------------------------------------------------------
00301 uint16_t EventFlag::GetMask()
00302 {
00303     // Return the presently held event flag values in this object.  Ensure
00304     // we get this within a critical section to guarantee atomicity.
00305     uint16_t u16Return;
00306     CS_ENTER();
00307     u16Return = m_u16SetMask;
00308     CS_EXIT();
00309     return u16Return;
00310 }
00311
00312 #endif // KERNEL_USE_EVENTFLAG
```

20.29 /home/moslevin/mark3-source/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 "autoalloc.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
#include "kerneldebug.h"
#include "tracebuffer.h"
```

20.29.1 Detailed Description

Kernel initialization and startup code.

Definition in file kernel.cpp.

20.30 kernel.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__   _|__  __|  __|_ _____
00004 |    \  /    | ||    \    |  ||      |    || |/ /    ||___  |
00005 |     \/     | ||     \    |  ||    __\    ||     \    ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007     |____|        |____|        |____|        |____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 #include "autoalloc.h"
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 PanicFunc_t Kernel::m_pfPanic;
00051
00052 #if KERNEL_USE_STACK_GUARD
00053 uint16_t Kernel::m_u16GuardThreshold;
00054 #endif
00055
00056 #if KERNEL_USE_IDLE_FUNC
00057 IdleFunc_t  Kernel::m_pfIdle;
00058 FakeThread_t Kernel::m_clIdle;
00059 #endif
00060
00061 #if KERNEL_USE_THREAD_CALLOUTS
00062 ThreadCreateCallout_t  Kernel::m_pfThreadCreateCallout;
00063 ThreadExitCallout_t    Kernel::m_pfThreadExitCallout;
00064 ThreadContextCallout_t Kernel::m_pfThreadContextCallout;
00065 #endif
00066 //---------------------------------------------------------------------------
00067 void Kernel::Init(void)
00068 {
00069 #if KERNEL_USE_AUTO_ALLOC
00070     AutoAlloc::Init();
00071 #endif
00072 #if KERNEL_USE_IDLE_FUNC
00073     ((Thread*)&m_clIdle)->InitIdle();
00074 #endif
00075 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00076     TraceBuffer::Init();
00077 #endif
00078     KERNEL_TRACE("Initializing Mark3 Kernel");
00079
00080     // Initialize the global kernel data - scheduler, timer-scheduler, and
00081     // the global message pool.
00082     Scheduler::Init();
00083 #if KERNEL_USE_DRIVER
00084     DriverList::Init();
00085 #endif
00086 #if KERNEL_USE_TIMERS
00087     TimerScheduler::Init();
00088 #endif
00089 #if KERNEL_USE_MESSAGE
00090     GlobalMessagePool::Init();
00091 #endif
00092 #if KERNEL_USE_PROFILER
00093     Profiler::Init();
00094 #endif
00095 #if KERNEL_USE_STACK_GUARD
00096     m_u16GuardThreshold = KERNEL_STACK_GUARD_DEFAULT;
00097 #endif
00098 }
00099
00100 //---------------------------------------------------------------------------
00101 void Kernel::Start(void)
00102 {
00103     KERNEL_TRACE("Starting Mark3 Scheduler");
00104     m_bIsStarted = true;
00105     ThreadPort::StartThreads();
00106     KERNEL_TRACE("Error starting Mark3 Scheduler");
00107 }
00108
```

```
00109 //---------------------------------------------------------------------
00110 void Kernel::Panic(uint16_t u16Cause_)
00111 {
00112     m_bIsPanic = true;
00113     if (m_pfPanic) {
00114         m_pfPanic(u16Cause_);
00115     } else {
00116 #if KERNEL_AWARE_SIMULATION
00117         KernelAware::Print("Panic\n");
00118         KernelAware::Trace(0, 0, u16Cause_, g_pclCurrent->
    GetID());
00119         KernelAware::ExitSimulator();
00120 #endif
00121         while (1)
00122             ;
00123     }
00124 }
```

20.31   /home/moslevin/mark3-source/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

    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.

20.31.1   Detailed Description

Kernel aware simulation support.

Definition in file kernelaware.cpp.

20.31.2   Variable Documentation

20.31.2.1   volatile bool g_bIsKernelAware

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

Definition at line 77 of file kernelaware.cpp.

20.31.2.2 KernelAwareData_t g_stKAData

Data structure used to communicate with host.

Definition at line 79 of file kernelaware.cpp.

20.32 kernelaware.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003 ___|   _|__   __|_       |__   __|_       |__   __| __   |__   _____
00004 |     \ /  |  | ||     \      | |         | |   |/ /      ||___    |
00005 |      \/   | | ||      \     | |     \    | |   \        ||__    |
00006 |__/\__/|__|__||__|\__\   __||__|\__\   __||__|\__\   __||_____|__|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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     volatile uint16_t au16Buffer[5];
00050
00054     struct {
00055         volatile const char* szName;
00056     } Profiler;
00061     struct {
00062         volatile uint16_t u16File;
00063         volatile uint16_t u16Line;
00064         volatile uint16_t u16Arg1;
00065         volatile uint16_t u16Arg2;
00066     } Trace;
00071     struct {
00072         volatile const char* szString;
00073     } Print;
00074 } KernelAwareData_t;
00075
00076 //---------------------------------------------------------------------------
00077 volatile bool      g_bIsKernelAware;
00078 volatile uint8_t   g_u8KACommand;
00079 KernelAwareData_t g_stKAData;
00080
00081 //---------------------------------------------------------------------------
00082 void KernelAware::ProfileInit(const char* szStr_)
00083 {
00084     CS_ENTER();
00085     g_stKAData.Profiler.szName = szStr_;
00086     g_u8KACommand              = KA_COMMAND_PROFILE_INIT;
00087     CS_EXIT();
00088 }
00089
00090 //---------------------------------------------------------------------------
00091 void KernelAware::ProfileStart(void)
00092 {
00093     g_u8KACommand = KA_COMMAND_PROFILE_START;
00094 }
00095
00096 //---------------------------------------------------------------------------
00097 void KernelAware::ProfileStop(void)
00098 {
00099     g_u8KACommand = KA_COMMAND_PROFILE_STOP;
00100 }
```

```
00101
00102 //---------------------------------------------------------------------------
00103 void KernelAware::ProfileReport(void)
00104 {
00105     g_u8KACommand = KA_COMMAND_PROFILE_REPORT;
00106 }
00107
00108 //---------------------------------------------------------------------------
00109 void KernelAware::ExitSimulator(void)
00110 {
00111     g_u8KACommand = KA_COMMAND_EXIT_SIMULATOR;
00112 }
00113
00114 //---------------------------------------------------------------------------
00115 void KernelAware::Trace(uint16_t u16File_, uint16_t u16Line_)
00116 {
00117     Trace_i(u16File_, u16Line_, 0, 0, KA_COMMAND_TRACE_0);
00118 }
00119
00120 //---------------------------------------------------------------------------
00121 void KernelAware::Trace(uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_)
00122 {
00123     Trace_i(u16File_, u16Line_, u16Arg1_, 0, KA_COMMAND_TRACE_1);
00124 }
00125 //---------------------------------------------------------------------------
00126 void KernelAware::Trace(uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t
     u16Arg2_)
00127 {
00128     Trace_i(u16File_, u16Line_, u16Arg1_, u16Arg2_, KA_COMMAND_TRACE_2);
00129 }
00130
00131 //---------------------------------------------------------------------------
00132 void KernelAware::Trace_i(
00133     uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_,
     KernelAwareCommand_t eCmd_)
00134 {
00135     CS_ENTER();
00136     g_stKAData.Trace.u16File = u16File_;
00137     g_stKAData.Trace.u16Line = u16Line_;
00138     g_stKAData.Trace.u16Arg1 = u16Arg1_;
00139     g_stKAData.Trace.u16Arg2 = u16Arg2_;
00140     g_u8KACommand           = eCmd_;
00141     CS_EXIT();
00142 }
00143
00144 //---------------------------------------------------------------------------
00145 void KernelAware::Print(const char* szStr_)
00146 {
00147     CS_ENTER();
00148     g_stKAData.Print.szString = szStr_;
00149     g_u8KACommand           = KA_COMMAND_PRINT;
00150     CS_EXIT();
00151 }
00152
00153 //---------------------------------------------------------------------------
00154 bool KernelAware::IsSimulatorAware(void)
00155 {
00156     return g_bIsKernelAware;
00157 }
00158
00159 #endif
```

20.33 /home/moslevin/mark3-source/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.

20.33.1   Detailed Description

Semaphore Blocking-Object Implemenation.

Definition in file ksemaphore.cpp.

20.33.2   Function Documentation

20.33.2.1   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 56 of file ksemaphore.cpp.

20.34   ksemaphore.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__|    _|__  __|_    |__|    _____
00004 |    \  /    |   | ||    \     ||     |    || |/ /     ||___    |
00005 |     \/     |   | ||     \    ||     |    || |  \      ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 #if KERNEL_USE_SEMAPHORE
00041
00042 #if KERNEL_USE_TIMEOUTS
00043 #include "timerlist.h"
00044
00045 //---------------------------------------------------------------------------
00056 void TimedSemaphore_Callback(Thread* pclOwner_, void* pvData_)
00057 {
00058     Semaphore* pclSemaphore = static_cast<Semaphore*>(pvData_);
00059
00060     // Indicate that the semaphore has expired on the thread
```

```
00061     pclOwner_->SetExpired(true);
00062
00063     // Wake up the thread that was blocked on this semaphore.
00064     pclSemaphore->WakeMe(pclOwner_);
00065
00066     if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
     ()->GetCurPriority()) {
00067         Thread::Yield();
00068     }
00069 }
00070
00071 //---------------------------------------------------------------------------
00072 Semaphore::~Semaphore()
00073 {
00074     // If there are any threads waiting on this object when it goes out
00075     // of scope, set a kernel panic.
00076     if (m_clBlockList.GetHead()) {
00077         Kernel::Panic(PANIC_ACTIVE_SEMAPHORE_DESCOPED);
00078     }
00079 }
00080
00081 //---------------------------------------------------------------------------
00082 void Semaphore::WakeMe(Thread* pclChosenOne_)
00083 {
00084     // Remove from the semaphore waitlist and back to its ready list.
00085     UnBlock(pclChosenOne_);
00086 }
00087
00088 #endif // KERNEL_USE_TIMEOUTS
00089
00090 //---------------------------------------------------------------------------
00091 uint8_t Semaphore::WakeNext()
00092 {
00093     Thread* pclChosenOne;
00094
00095     pclChosenOne = m_clBlockList.HighestWaiter();
00096
00097     // Remove from the semaphore waitlist and back to its ready list.
00098     UnBlock(pclChosenOne);
00099
00100     // Call a task switch if higher or equal priority thread
00101     if (pclChosenOne->GetCurPriority() >=
     Scheduler::GetCurrentThread()->GetCurPriority()) {
00102         return 1;
00103     }
00104     return 0;
00105 }
00106
00107 //---------------------------------------------------------------------------
00108 void Semaphore::Init(uint16_t u16InitVal_, uint16_t u16MaxVal_)
00109 {
00110     // Copy the paramters into the object - set the maximum value for this
00111     // semaphore to implement either binary or counting semaphores, and set
00112     // the initial count.  Clear the wait list for this object.
00113     m_u16Value    = u16InitVal_;
00114     m_u16MaxValue = u16MaxVal_;
00115
00116     m_clBlockList.Init();
00117 }
00118
00119 //---------------------------------------------------------------------------
00120 bool Semaphore::Post()
00121 {
00122     KERNEL_TRACE_1("Posting semaphore, Thread %d", (uint16_t)
     g_pclCurrent->GetID());
00123
00124     bool bThreadWake = 0;
00125     bool bBail       = false;
00126     // Increment the semaphore count - we can mess with threads so ensure this
00127     // is in a critical section.  We don't just disable the scheudler since
00128     // we want to be able to do this from within an interrupt context as well.
00129     CS_ENTER();
00130
00131     // If nothing is waiting for the semaphore
00132     if (m_clBlockList.GetHead() == NULL) {
00133         // Check so see if we've reached the maximum value in the semaphore
00134         if (m_u16Value < m_u16MaxValue) {
00135             // Increment the count value
00136             m_u16Value++;
00137         } else {
00138             // Maximum value has been reached, bail out.
00139             bBail = true;
00140         }
00141     } else {
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            return false;
00152        }
00153
00154        // if bThreadWake was set, it means that a higher-priority thread was
00155        // woken.  Trigger a context switch to ensure that this thread gets
00156        // to execute next.
00157        if (bThreadWake) {
00158            Thread::Yield();
00159        }
00160        return true;
00161 }
00162
00163 //---------------------------------------------------------------------------
00164 #if KERNEL_USE_TIMEOUTS
00165 bool Semaphore::Pend_i(uint32_t u32WaitTimeMS_)
00166 #else
00167 void Semaphore::Pend_i(void)
00168 #endif
00169 {
00170        KERNEL_TRACE_1("Pending semaphore, Thread %d", (uint16_t)
       g_pclCurrent->GetID());
00171
00172 #if KERNEL_USE_TIMEOUTS
00173        Timer clSemTimer;
00174        bool  bUseTimer = false;
00175 #endif
00176
00177        // Once again, messing with thread data - ensure
00178        // we're doing all of these operations from within a thread-safe context.
00179        CS_ENTER();
00180
00181        // Check to see if we need to take any action based on the semaphore count
00182        if (m_u16Value != 0) {
00183            // The semaphore count is non-zero, we can just decrement the count
00184            // and go along our merry way.
00185            m_u16Value--;
00186        } else {
00187 // The semaphore count is zero - we need to block the current thread
00188 // and wait until the semaphore is posted from elsewhere.
00189 #if KERNEL_USE_TIMEOUTS
00190            if (u32WaitTimeMS_) {
00191                g_pclCurrent->SetExpired(false);
00192                clSemTimer.Init();
00193                clSemTimer.Start(0, u32WaitTimeMS_, TimedSemaphore_Callback, (void*)this
       );
00194                bUseTimer = true;
00195            }
00196 #endif
00197            BlockPriority(g_pclCurrent);
00198
00199            // Switch Threads immediately
00200            Thread::Yield();
00201        }
00202
00203        CS_EXIT();
00204
00205 #if KERNEL_USE_TIMEOUTS
00206        if (bUseTimer) {
00207            clSemTimer.Stop();
00208            return (g_pclCurrent->GetExpired() == 0);
00209        }
00210        return true;
00211 #endif
00212 }
00213
00214 //---------------------------------------------------------------------------
00215 // Redirect the untimed pend API to the timed pend, with a null timeout.
00216 void Semaphore::Pend()
00217 {
00218 #if KERNEL_USE_TIMEOUTS
00219        Pend_i(0);
00220 #else
00221        Pend_i();
00222 #endif
00223 }
00224
00225 #if KERNEL_USE_TIMEOUTS
00226 //---------------------------------------------------------------------------
00227 bool Semaphore::Pend(uint32_t u32WaitTimeMS_)
00228 {
00229        return Pend_i(u32WaitTimeMS_);
```

```
00230 }
00231 #endif
00232
00233 //---------------------------------------------------------------------------
00234 uint16_t Semaphore::GetCount()
00235 {
00236     uint16_t u16Ret;
00237     CS_ENTER();
00238     u16Ret = m_u16Value;
00239     CS_EXIT();
00240     return u16Ret;
00241 }
00242
00243 #endif
```

20.35   /home/moslevin/mark3-source/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"
```

20.35.1   Detailed Description

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

Definition in file ll.cpp.

20.36   ll.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__    __|_ |__    __|__ __|__   |__ |_____|____
00004 |    \  /  |  |  | ||    \      ||    |     || / /     ||___    |
00005 |     \/   | ||    \     ||      \      ||     \        ||___    |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||___|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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     node_->prev = m_pstTail;
00052     node_->next = NULL;
00053
00054     // If the list is empty, initilize the head
00055     if (!m_pstHead) {
00056         m_pstHead = node_;
00057     }
00058     // Otherwise, adjust the tail's next pointer
00059     else {
00060         m_pstTail->next = node_;
00061     }
00062
00063     // Move the tail node, and assign it to the new node just passed in
00064     m_pstTail = node_;
00065 }
00066
00067 //---------------------------------------------------------------------------
00068 void DoubleLinkList::Remove(LinkListNode* node_)
00069 {
00070     KERNEL_ASSERT(node_);
00071
00072     if (node_->prev) {
00073 #if SAFE_UNLINK
00074         if (node_->prev->next != node_) {
00075             Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00076         }
00077 #endif
00078         node_->prev->next = node_->next;
00079     }
00080     if (node_->next) {
00081 #if SAFE_UNLINK
00082         if (node_->next->prev != node_) {
00083             Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00084         }
00085 #endif
00086         node_->next->prev = node_->prev;
00087     }
00088     if (node_ == m_pstHead) {
00089         m_pstHead = node_->next;
00090     }
00091     if (node_ == m_pstTail) {
00092         m_pstTail = node_->prev;
00093     }
00094 }
00095
00096 //---------------------------------------------------------------------------
00097 void CircularLinkList::Add(LinkListNode* node_)
00098 {
00099     KERNEL_ASSERT(node_);
00100
00101     if (!m_pstHead) {
00102         // If the list is empty, initilize the nodes
00103         m_pstHead = node_;
00104         m_pstTail = node_;
00105     } else {
00106         // Move the tail node, and assign it to the new node just passed in
00107         m_pstTail->next = node_;
00108     }
00109
00110     // Add a node to the end of the linked list.
00111     node_->prev = m_pstTail;
00112     node_->next = m_pstHead;
00113
00114     m_pstTail       = node_;
00115     m_pstHead->prev = node_;
00116 }
00117
00118 //---------------------------------------------------------------------------
00119 void CircularLinkList::Remove(LinkListNode* node_)
00120 {
00121     KERNEL_ASSERT(node_);
00122
00123     // Check to see if this is the head of the list...
00124     if ((node_ == m_pstHead) && (m_pstHead == m_pstTail)) {
00125         // Clear the head and tail pointers - nothing else left.
00126         m_pstHead = NULL;
00127         m_pstTail = NULL;
00128         return;
00129     }
00130
00131 #if SAFE_UNLINK
00132     // Verify that all nodes are properly connected
```

```
00133      if ((node_->prev->next != node_) || (node_->next->prev != node_)) {
00134          Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
00135      }
00136 #endif
00137
00138      // This is a circularly linked list - no need to check for connection,
00139      // just remove the node.
00140      node_->next->prev = node_->prev;
00141      node_->prev->next = node_->next;
00142
00143      if (node_ == m_pstHead) {
00144          m_pstHead = m_pstHead->next;
00145      }
00146      if (node_ == m_pstTail) {
00147          m_pstTail = m_pstTail->prev;
00148      }
00149      node_->ClearNode();
00150 }
00151
00152 //---------------------------------------------------------------------------
00153 void CircularLinkList::PivotForward()
00154 {
00155      if (m_pstHead) {
00156          m_pstHead = m_pstHead->next;
00157          m_pstTail = m_pstTail->next;
00158      }
00159 }
00160
00161 //---------------------------------------------------------------------------
00162 void CircularLinkList::PivotBackward()
00163 {
00164      if (m_pstHead) {
00165          m_pstHead = m_pstHead->prev;
00166          m_pstTail = m_pstTail->prev;
00167      }
00168 }
00169
00170 //---------------------------------------------------------------------------
00171 void CircularLinkList::InsertNodeBefore(
00172 LinkListNode* node_, LinkListNode* insert_)
00172 {
00173      KERNEL_ASSERT(node_);
00174
00175      node_->next = insert_;
00176      node_->prev = insert_->prev;
00177
00178      if (insert_->prev) {
00179          insert_->prev->next = node_;
00180      }
00181      insert_->prev = node_;
00182 }
```

20.37 /home/moslevin/mark3-source/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"
```

20.37.1 Detailed Description

Mailbox + Envelope IPC mechanism.

Definition in file mailbox.cpp.

20.38   mailbox.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  __|  _____|
00004 |    \  /  | ||    \    ||    |    ||    ||  1/ /     ||___    |
00005 |     \/   | ||     \    ||    |    ||    ||  \       ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 Mailbox::~Mailbox()
00043 {
00044     // If the mailbox isn't empty on destruction, kernel panic.
00045     if (m_u16Free != m_u16Count) {
00046         Kernel::Panic(PANIC_ACTIVE_MAILBOX_DESCOPED);
00047     }
00048 }
00049
00050 //---------------------------------------------------------------------------
00051 void Mailbox::Init(void* pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_)
00052 {
00053     KERNEL_ASSERT(u16BufferSize_);
00054     KERNEL_ASSERT(u16ElementSize_);
00055     KERNEL_ASSERT(pvBuffer_);
00056
00057     m_pvBuffer      = pvBuffer_;
00058     m_u16ElementSize = u16ElementSize_;
00059
00060     m_u16Count = (u16BufferSize_ / u16ElementSize_);
00061     m_u16Free  = m_u16Count;
00062
00063     m_u16Head = 0;
00064     m_u16Tail = 0;
00065
00066     // We use the counting semaphore to implement blocking - with one element
00067     // in the mailbox corresponding to a post/pend operation in the semaphore.
00068     m_clRecvSem.Init(0, m_u16Free);
00069
00070 #if KERNEL_USE_TIMEOUTS
00071     // Binary semaphore is used to track any threads that are blocked on a
00072     // "send" due to lack of free slots.
00073     m_clSendSem.Init(0, 1);
00074 #endif
00075 }
00076
00077 //---------------------------------------------------------------------------
00078 #if KERNEL_USE_AUTO_ALLOC
00079 Mailbox* Mailbox::Init(uint16_t u16BufferSize_, uint16_t u16ElementSize_)
00080 {
00081     Mailbox* pclNew   = (Mailbox*)AutoAlloc::Allocate(sizeof(
     Mailbox));
00082     void*    pvBuffer = AutoAlloc::Allocate(u16BufferSize_);
00083     pclNew->Init(pvBuffer, u16BufferSize_, u16ElementSize_);
00084     return pclNew;
00085 }
00086 #endif
00087
00088 //---------------------------------------------------------------------------
00089 void Mailbox::Receive(void* pvData_)
00090 {
```

```
00091       KERNEL_ASSERT(pvData_);
00092
00093 #if KERNEL_USE_TIMEOUTS
00094       Receive_i(pvData_, false, 0);
00095 #else
00096       Receive_i(pvData_, false);
00097 #endif
00098 }
00099
00100 #if KERNEL_USE_TIMEOUTS
00101 //---------------------------------------------------------------------------
00102 bool Mailbox::Receive(void* pvData_, uint32_t u32TimeoutMS_)
00103 {
00104       KERNEL_ASSERT(pvData_);
00105       return Receive_i(pvData_, false, u32TimeoutMS_);
00106 }
00107 #endif
00108
00109 //---------------------------------------------------------------------------
00110 void Mailbox::ReceiveTail(void* pvData_)
00111 {
00112       KERNEL_ASSERT(pvData_);
00113
00114 #if KERNEL_USE_TIMEOUTS
00115       Receive_i(pvData_, true, 0);
00116 #else
00117       Receive_i(pvData_, true);
00118 #endif
00119 }
00120
00121 #if KERNEL_USE_TIMEOUTS
00122 //---------------------------------------------------------------------------
00123 bool Mailbox::ReceiveTail(void* pvData_, uint32_t u32TimeoutMS_)
00124 {
00125       KERNEL_ASSERT(pvData_);
00126       return Receive_i(pvData_, true, u32TimeoutMS_);
00127 }
00128 #endif
00129
00130 //---------------------------------------------------------------------------
00131 bool Mailbox::Send(void* pvData_)
00132 {
00133       KERNEL_ASSERT(pvData_);
00134
00135 #if KERNEL_USE_TIMEOUTS
00136       return Send_i(pvData_, false, 0);
00137 #else
00138       return Send_i(pvData_, false);
00139 #endif
00140 }
00141
00142 //---------------------------------------------------------------------------
00143 bool Mailbox::SendTail(void* pvData_)
00144 {
00145       KERNEL_ASSERT(pvData_);
00146
00147 #if KERNEL_USE_TIMEOUTS
00148       return Send_i(pvData_, true, 0);
00149 #else
00150       return Send_i(pvData_, true);
00151 #endif
00152 }
00153
00154 #if KERNEL_USE_TIMEOUTS
00155 //---------------------------------------------------------------------------
00156 bool Mailbox::Send(void* pvData_, uint32_t u32TimeoutMS_)
00157 {
00158       KERNEL_ASSERT(pvData_);
00159
00160       return Send_i(pvData_, false, u32TimeoutMS_);
00161 }
00162
00163 //---------------------------------------------------------------------------
00164 bool Mailbox::SendTail(void* pvData_, uint32_t u32TimeoutMS_)
00165 {
00166       KERNEL_ASSERT(pvData_);
00167
00168       return Send_i(pvData_, true, u32TimeoutMS_);
00169 }
00170 #endif
00171
00172 //---------------------------------------------------------------------------
00173 #if KERNEL_USE_TIMEOUTS
00174 bool Mailbox::Send_i(const void* pvData_, bool bTail_, uint32_t u32TimeoutMS_)
00175 #else
00176 bool Mailbox::Send_i(const void* pvData_, bool bTail_)
00177 #endif
```

```
00178 {
00179     const void* pvDst;
00180
00181     bool bRet      = false;
00182     bool bSchedState = Scheduler::SetScheduler(false);
00183
00184 #if KERNEL_USE_TIMEOUTS
00185     bool bBlock = false;
00186     bool bDone  = false;
00187     while (!bDone) {
00188         // Try to claim a slot first before resorting to blocking.
00189         if (bBlock) {
00190             bDone = true;
00191             Scheduler::SetScheduler(bSchedState);
00192             m_clSendSem.Pend(u32TimeoutMS_);
00193             Scheduler::SetScheduler(false);
00194         }
00195 #endif
00196
00197         CS_ENTER();
00198         // Ensure we have a free slot before we attempt to write data
00199         if (m_u16Free) {
00200             m_u16Free--;
00201
00202             if (bTail_) {
00203                 pvDst = GetTailPointer();
00204                 MoveTailBackward();
00205             } else {
00206                 MoveHeadForward();
00207                 pvDst = GetHeadPointer();
00208             }
00209             bRet = true;
00210 #if KERNEL_USE_TIMEOUTS
00211             bDone = true;
00212 #endif
00213         }
00214 #if KERNEL_USE_TIMEOUTS
00215         else if (u32TimeoutMS_) {
00216             bBlock = true;
00217         } else {
00218             bDone = true;
00219         }
00220 #endif
00221
00222         CS_EXIT();
00223
00224 #if KERNEL_USE_TIMEOUTS
00225     }
00226 #endif
00227
00228     // Copy data to the claimed slot, and post the counting semaphore
00229     if (bRet) {
00230         CopyData(pvData_, pvDst, m_u16ElementSize);
00231     }
00232
00233     Scheduler::SetScheduler(bSchedState);
00234
00235     if (bRet) {
00236         m_clRecvSem.Post();
00237     }
00238
00239     return bRet;
00240 }
00241
00242 //---------------------------------------------------------------------------
00243 #if KERNEL_USE_TIMEOUTS
00244 bool Mailbox::Receive_i(const void* pvData_, bool bTail_, uint32_t u32WaitTimeMS_)
00245 #else
00246 void Mailbox::Receive_i(const void* pvData_, bool bTail_)
00247 #endif
00248 {
00249     const void* pvSrc;
00250
00251 #if KERNEL_USE_TIMEOUTS
00252     if (!m_clRecvSem.Pend(u32WaitTimeMS_)) {
00253         // Failed to get the notification from the counting semaphore in the
00254         // time allotted.  Bail.
00255         return false;
00256     }
00257 #else
00258     m_clRecvSem.Pend();
00259 #endif
00260
00261     // Disable the scheduler while we do this -- this ensures we don't have
00262     // multiple concurrent readers off the same queue, which could be problematic
00263     // if multiple writes occur during reads, etc.
00264     bool bSchedState = Scheduler::SetScheduler(false);
```

```
00265
00266        // Update the head/tail indexes, and get the associated data pointer for
00267        // the read operation.
00268        CS_ENTER();
00269
00270        m_u16Free++;
00271        if (bTail_) {
00272            MoveTailForward();
00273            pvSrc = GetTailPointer();
00274        } else {
00275            pvSrc = GetHeadPointer();
00276            MoveHeadBackward();
00277        }
00278
00279        CS_EXIT();
00280
00281        CopyData(pvSrc, pvData_, m_u16ElementSize);
00282
00283        Scheduler::SetScheduler(bSchedState);
00284
00285 #if KERNEL_USE_TIMEOUTS
00286        // Unblock a thread waiting for a free slot to send to
00287        m_clSendSem.Post();
00288
00289        return true;
00290 #endif
00291 }
00292
00293 #endif
```

20.39   /home/moslevin/mark3-source/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"
```

20.39.1   Detailed Description

Inter-thread communications via message passing.

Definition in file message.cpp.

20.40   message.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /    |  ||    \    ||    |      ||    |/ /      ||___   |
00005 |     \/     |  ||     \    ||     \    ||    |\    \    ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 MessagePool GlobalMessagePool::m_clPool;
00048
00049 //---------------------------------------------------------------------------
00050 void MessagePool::Init()
00051 {
00052     m_clList.Init();
00053 }
00054
00055 //---------------------------------------------------------------------------
00056 void MessagePool::Push(Message* pclMessage_)
00057 {
00058     KERNEL_ASSERT(pclMessage_);
00059
00060     CS_ENTER();
00061
00062     m_clList.Add(pclMessage_);
00063
00064     CS_EXIT();
00065 }
00066
00067 //---------------------------------------------------------------------------
00068 Message* MessagePool::Pop()
00069 {
00070     Message* pclRet;
00071     CS_ENTER();
00072
00073     pclRet = static_cast<Message*>(m_clList.GetHead());
00074     if (0 != pclRet) {
00075         m_clList.Remove(static_cast<LinkListNode*>(pclRet));
00076     }
00077
00078     CS_EXIT();
00079     return pclRet;
00080 }
00081
00082 //---------------------------------------------------------------------------
00083 Message* MessagePool::GetHead()
00084 {
00085     return static_cast<Message*>(m_clList.GetHead());
00086 }
00087
00088 //---------------------------------------------------------------------------
00089 void GlobalMessagePool::Init()
00090 {
00091     uint8_t i;
00092     GlobalMessagePool::m_clPool.Init();
00093     for (i = 0; i < GLOBAL_MESSAGE_POOL_SIZE; i++) {
00094         GlobalMessagePool::m_aclMessagePool[i].Init();
00095         GlobalMessagePool::m_clPool.Push(&(GlobalMessagePool::m_aclMessagePool[i]));
00096     }
00097 }
00098
00099 //---------------------------------------------------------------------------
00100 void GlobalMessagePool::Push(Message* pclMessage_)
00101 {
00102     m_clPool.Push(pclMessage_);
00103 }
00104
00105 //---------------------------------------------------------------------------
00106 Message* GlobalMessagePool::Pop()
00107 {
00108     return m_clPool.Pop();
00109 }
00110
00111 //---------------------------------------------------------------------------
00112 Message* GlobalMessagePool::GetHead()
00113 {
```

```
00114     return m_clPool.GetHead();
00115 }
00116
00117 //---------------------------------------------------------------------------
00118 MessagePool* GlobalMessagePool::GetPool()
00119 {
00120     return &m_clPool;
00121 }
00122
00123 //---------------------------------------------------------------------------
00124 void MessageQueue::Init()
00125 {
00126     m_clSemaphore.Init(0, GLOBAL_MESSAGE_POOL_SIZE);
00127 }
00128
00129 //---------------------------------------------------------------------------
00130 Message* MessageQueue::Receive()
00131 {
00132 #if KERNEL_USE_TIMEOUTS
00133     return Receive_i(0);
00134 #else
00135     return Receive_i();
00136 #endif
00137 }
00138
00139 //---------------------------------------------------------------------------
00140 #if KERNEL_USE_TIMEOUTS
00141 Message* MessageQueue::Receive(uint32_t u32TimeWaitMS_)
00142 {
00143     return Receive_i(u32TimeWaitMS_);
00144 }
00145 #endif
00146
00147 //---------------------------------------------------------------------------
00148 #if KERNEL_USE_TIMEOUTS
00149 Message* MessageQueue::Receive_i(uint32_t u32TimeWaitMS_)
00150 #else
00151 Message* MessageQueue::Receive_i(void)
00152 #endif
00153 {
00154     Message* pclRet;
00155
00156 // Block the current thread on the counting semaphore
00157 #if KERNEL_USE_TIMEOUTS
00158     if (!m_clSemaphore.Pend(u32TimeWaitMS_)) {
00159         return NULL;
00160     }
00161 #else
00162     m_clSemaphore.Pend();
00163 #endif
00164
00165     CS_ENTER();
00166
00167     // Pop the head of the message queue and return it
00168     pclRet = static_cast<Message*>(m_clLinkList.GetHead());
00169     m_clLinkList.Remove(static_cast<Message*>(pclRet));
00170
00171     CS_EXIT();
00172
00173     return pclRet;
00174 }
00175
00176 //---------------------------------------------------------------------------
00177 void MessageQueue::Send(Message* pclSrc_)
00178 {
00179     KERNEL_ASSERT(pclSrc_);
00180
00181     CS_ENTER();
00182
00183     // Add the message to the head of the linked list
00184     m_clLinkList.Add(pclSrc_);
00185
00186     // Post the semaphore, waking the blocking thread for the queue.
00187     m_clSemaphore.Post();
00188
00189     CS_EXIT();
00190 }
00191
00192 //---------------------------------------------------------------------------
00193 uint16_t MessageQueue::GetCount()
00194 {
00195     return m_clSemaphore.GetCount();
00196 }
00197 #endif // KERNEL_USE_MESSAGE
```

20.41 /home/moslevin/mark3-source/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.

20.41.1 Detailed Description

Mutual-exclusion object.

Definition in file mutex.cpp.

20.41.2 Function Documentation

20.41.2.1 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.

20.42 mutex.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__   __|__    |__   _____
00004 |    \  /    |  |  ||      \       ||      |       ||  |/ /        ||___   |
00005 |     \/     |  |  ||       \      ||      \       ||  |  / /      ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\     ||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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         Thread::Yield();
00066     }
00067 }
00068 //----------------------------------------------------------------------------
00069 Mutex::~Mutex()
00070 {
00071     // If there are any threads waiting on this object when it goes out
00072     // of scope, set a kernel panic.
00073     if (m_clBlockList.GetHead()) {
00074         Kernel::Panic(PANIC_ACTIVE_MUTEX_DESCOPED);
00075     }
00076 }
00077
00078 //----------------------------------------------------------------------------
00079 void Mutex::WakeMe(Thread* pclOwner_)
00080 {
00081     // Remove from the semaphore waitlist and back to its ready list.
00082     UnBlock(pclOwner_);
00083 }
00084
00085 #endif
00086
00087 //----------------------------------------------------------------------------
00088 uint8_t Mutex::WakeNext()
00089 {
00090     Thread* pclChosenOne = NULL;
00091
00092     // Get the highest priority waiter thread
00093     pclChosenOne = m_clBlockList.HighestWaiter();
00094
00095     // Unblock the thread
00096     UnBlock(pclChosenOne);
00097
00098     // The chosen one now owns the mutex
00099     m_pclOwner = pclChosenOne;
00100
00101     // Signal a context switch if it's a greater than or equal to the current priority
00102     if (pclChosenOne->GetCurPriority() >=
    Scheduler::GetCurrentThread()->GetCurPriority()) {
00103         return 1;
00104     }
00105     return 0;
00106 }
00107
00108 //----------------------------------------------------------------------------
00109 void Mutex::Init()
00110 {
00111     // Reset the data in the mutex
00112     m_bReady   = 1;    // The mutex is free.
00113     m_u8MaxPri = 0;    // Set the maximum priority inheritence state
00114     m_pclOwner = NULL; // Clear the mutex owner
00115     m_u8Recurse = 0;   // Reset recurse count
00116 }
00117
00118 //----------------------------------------------------------------------------
00119 #if KERNEL_USE_TIMEOUTS
00120 bool Mutex::Claim_i(uint32_t u32WaitTimeMS_)
00121 #else
00122 void Mutex::Claim_i(void)
```

```
00123 #endif
00124 {
00125     KERNEL_TRACE_1("Claiming Mutex, Thread %d", (uint16_t)
      g_pclCurrent->GetID());
00126
00127 #if KERNEL_USE_TIMEOUTS
00128     Timer clTimer;
00129     bool  bUseTimer = false;
00130 #endif
00131
00132     // Disable the scheduler while claiming the mutex - we're dealing with all
00133     // sorts of private thread data, can't have a thread switch while messing
00134     // with internal data structures.
00135     Scheduler::SetScheduler(0);
00136
00137     // Check to see if the mutex is claimed or not
00138     if (m_bReady != 0) {
00139         // Mutex isn't claimed, claim it.
00140         m_bReady    = 0;
00141         m_u8Recurse = 0;
00142         m_u8MaxPri  = g_pclCurrent->GetPriority();
00143         m_pclOwner  = g_pclCurrent;
00144
00145         Scheduler::SetScheduler(1);
00146
00147 #if KERNEL_USE_TIMEOUTS
00148         return true;
00149 #else
00150         return;
00151 #endif
00152     }
00153
00154     // If the mutex is already claimed, check to see if this is the owner thread,
00155     // since we allow the mutex to be claimed recursively.
00156     if (g_pclCurrent == m_pclOwner) {
00157         // Ensure that we haven't exceeded the maximum recursive-lock count
00158         KERNEL_ASSERT((m_u8Recurse < 255));
00159         m_u8Recurse++;
00160
00161         // Increment the lock count and bail
00162         Scheduler::SetScheduler(1);
00163 #if KERNEL_USE_TIMEOUTS
00164         return true;
00165 #else
00166         return;
00167 #endif
00168     }
00169
00170 // The mutex is claimed already - we have to block now.  Move the
00171 // current thread to the list of threads waiting on the mutex.
00172 #if KERNEL_USE_TIMEOUTS
00173     if (u32WaitTimeMS_) {
00174         g_pclCurrent->SetExpired(false);
00175         clTimer.Init();
00176         clTimer.Start(0, u32WaitTimeMS_, (TimerCallback_t)
      TimedMutex_Calback, (void*)this);
00177         bUseTimer = true;
00178     }
00179 #endif
00180     BlockPriority(g_pclCurrent);
00181
00182     // Check if priority inheritence is necessary.  We do this in order
00183     // to ensure that we don't end up with priority inversions in case
00184     // multiple threads are waiting on the same resource.
00185     if (m_u8MaxPri <= g_pclCurrent->GetPriority()) {
00186         m_u8MaxPri = g_pclCurrent->GetPriority();
00187
00188         Thread* pclTemp = static_cast<Thread*>(m_clBlockList.GetHead());
00189         while (pclTemp) {
00190             pclTemp->InheritPriority(m_u8MaxPri);
00191             if (pclTemp == static_cast<Thread*>(m_clBlockList.GetTail())) {
00192                 break;
00193             }
00194             pclTemp = static_cast<Thread*>(pclTemp->GetNext());
00195         }
00196         m_pclOwner->InheritPriority(m_u8MaxPri);
00197     }
00198
00199     // Done with thread data -reenable the scheduler
00200     Scheduler::SetScheduler(1);
00201
00202     // Switch threads if this thread acquired the mutex
00203     Thread::Yield();
00204
00205 #if KERNEL_USE_TIMEOUTS
00206     if (bUseTimer) {
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            m_u8Recurse--;
00249            Scheduler::SetScheduler(1);
00250            return;
00251        }
00252
00253        // Restore the thread's original priority
00254        if (g_pclCurrent->GetCurPriority() != g_pclCurrent->
      GetPriority()) {
00255            g_pclCurrent->SetPriority(g_pclCurrent->
      GetPriority());
00256
00257            // In this case, we want to reschedule
00258            bSchedule = 1;
00259        }
00260
00261        // No threads are waiting on this semaphore?
00262        if (m_clBlockList.GetHead() == NULL) {
00263            // Re-initialize the mutex to its default values
00264            m_bReady    = 1;
00265            m_u8MaxPri = 0;
00266            m_pclOwner = NULL;
00267        } else {
00268            // Wake the highest priority Thread pending on the mutex
00269            if (WakeNext()) {
00270                // Switch threads if it's higher or equal priority than the current thread
00271                bSchedule = 1;
00272            }
00273        }
00274
00275        // Must enable the scheduler again in order to switch threads.
00276        Scheduler::SetScheduler(1);
00277        if (bSchedule) {
00278            // Switch threads if a higher-priority thread was woken
00279            Thread::Yield();
00280        }
00281 }
00282
00283 #endif // KERNEL_USE_MUTEX
```

20.43 /home/moslevin/mark3-source/embedded/kernel/notify.cpp File Reference

Lightweight thread notification - blocking object.

```
#include "mark3cfg.h"
#include "notify.h"
#include "mark3.h"
#include "dbg_file_list.h"
#include "buffalogger.h"
```

20.43.1 Detailed Description

Lightweight thread notification - blocking object.

Definition in file notify.cpp.

20.44 notify.cpp

```
00001 /*===========================================================================
00002       _____        _____        _____        _____
00003 ___|    _|__   __|_     |__    __|__    |__    __|  __    |__  _____
00004 |     \ /   |  | ||      \      | |      |       | |/ /      | |___    |
00005 |      \/   |  | ||       \     | |      |       | |         | |___    |
00006 |__/\__/ |__|_||__|\__\   _||__|\__\   _||__|\__\   _||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #include "mark3cfg.h"
00023 #include "notify.h"
00024 #include "mark3.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 #if KERNEL_USE_TIMEOUTS
00040 //---------------------------------------------------------------------------
00041 void TimedNotify_Callback(Thread* pclOwner_, void* pvData_)
00042 {
00043     Notify* pclNotify = static_cast<Notify*>(pvData_);
00044
00045     // Indicate that the semaphore has expired on the thread
00046     pclOwner_->SetExpired(true);
00047
00048     // Wake up the thread that was blocked on this semaphore.
00049     pclNotify->WakeMe(pclOwner_);
00050
00051     if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread
00052 ()->GetCurPriority()) {
00052         Thread::Yield();
00053     }
00054 }
00055 #endif
00056 //---------------------------------------------------------------------------
00057 Notify::~Notify()
00058 {
00059     // If there are any threads waiting on this object when it goes out
00060     // of scope, set a kernel panic.
00061     if (m_clBlockList.GetHead()) {
00062         Kernel::Panic(PANIC_ACTIVE_NOTIFY_DESCOPED);
00063     }
00064 }
00065
00066 //---------------------------------------------------------------------------
00067 void Notify::Init(void)
00068 {
00069     m_clBlockList.Init();
```

```
00070 }
00071
00072 //---------------------------------------------------------------------------
00073 void Notify::Signal(void)
00074 {
00075     bool bReschedule = false;
00076
00077     CS_ENTER();
00078     Thread* pclCurrent = (Thread*)m_clBlockList.GetHead();
00079     while (pclCurrent != NULL) {
00080         UnBlock(pclCurrent);
00081         if (!bReschedule && (pclCurrent->GetCurPriority() >=
     Scheduler::GetCurrentThread()->GetCurPriority())) {
00082             bReschedule = true;
00083         }
00084         pclCurrent = (Thread*)m_clBlockList.GetHead();
00085     }
00086     CS_EXIT();
00087
00088     if (bReschedule) {
00089         Thread::Yield();
00090     }
00091 }
00092
00093 //---------------------------------------------------------------------------
00094 void Notify::Wait(bool* pbFlag_)
00095 {
00096     CS_ENTER();
00097     Block(g_pclCurrent);
00098     if (pbFlag_) {
00099         *pbFlag_ = false;
00100     }
00101     CS_EXIT();
00102
00103     Thread::Yield();
00104     if (pbFlag_) {
00105         *pbFlag_ = true;
00106     }
00107 }
00108
00109 //---------------------------------------------------------------------------
00110 #if KERNEL_USE_TIMEOUTS
00111 bool Notify::Wait(uint32_t u32WaitTimeMS_, bool* pbFlag_)
00112 {
00113     bool  bUseTimer = false;
00114     Timer clNotifyTimer;
00115
00116     CS_ENTER();
00117     if (u32WaitTimeMS_) {
00118         bUseTimer = true;
00119         g_pclCurrent->SetExpired(false);
00120
00121         clNotifyTimer.Init();
00122         clNotifyTimer.Start(0, u32WaitTimeMS_, TimedNotify_Callback, (void*)this);
00123     }
00124
00125     Block(g_pclCurrent);
00126
00127     if (pbFlag_) {
00128         *pbFlag_ = false;
00129     }
00130     CS_EXIT();
00131
00132     Thread::Yield();
00133
00134     if (bUseTimer) {
00135         clNotifyTimer.Stop();
00136         return (g_pclCurrent->GetExpired() == 0);
00137     }
00138
00139     if (pbFlag_) {
00140         *pbFlag_ = true;
00141     }
00142
00143     return true;
00144 }
00145 #endif
00146 //---------------------------------------------------------------------------
00147 void Notify::WakeMe(Thread* pclChosenOne_)
00148 {
00149     UnBlock(pclChosenOne_);
00150 }
00151
00152 #endif
```

20.45 /home/moslevin/mark3-source/embedded/kernel/priomap.cpp File Reference

Priority map data structure.

```
#include "mark3.h"
#include "priomap.h"
#include "threadport.h"
#include <stdint.h>
#include <stdbool.h>
```

20.45.1 Detailed Description

Priority map data structure.

Definition in file priomap.cpp.

20.46 priomap.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____|  __|__
00004 |    \  /   | ||    \      ||      |      ||   |/ /      ||___  |
00005 |     \/    | ||     \     ||     _\     ||   |\ \       ||___  |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007     |_____|      |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00019 #include "mark3.h"
00020 #include "priomap.h"
00021 #include "threadport.h"
00022
00023 #include <stdint.h>
00024 #include <stdbool.h>
00025
00026 //---------------------------------------------------------------------------
00027 static inline uint8_t priority_from_bitmap(PORT_PRIO_TYPE uXPrio_)
00028 {
00029 #if HW_CLZ
00030     // Support hardware-accelerated Count-leading-zeros instruction
00031     uint8_t rc = PRIO_MAP_BITS - CLZ(uXPrio_);
00032     return rc;
00033 #else
00034     // Default un-optimized count-leading zeros operation
00035     PORT_PRIO_TYPE uXMask  = (1 << (PRIO_MAP_BITS - 1));
00036     uint8_t    u8Zeros = 0;
00037
00038     while (uXMask) {
00039         if (uXMask & uXPrio_) {
00040             return (PRIO_MAP_BITS - u8Zeros);
00041         }
00042
00043         uXMask >>= 1;
00044         u8Zeros++;
00045     }
00046     return 0;
00047 #endif
00048 }
00049
00050 //---------------------------------------------------------------------------
00051 PriorityMap::PriorityMap()
00052 {
00053 #if PRIO_MAP_MULTI_LEVEL
00054     m_uXPriorityMapL2 = 0;
00055     for (int i = 0; i < PRIO_MAP_NUM_WORDS; i++) {
00056         m_auXPriorityMap[i] = 0;
00057     }
00058 #else
00059     m_uXPriorityMap = 0;
00060 #endif
00061 }
```

```
00062
00063 //---------------------------------------------------------------------------
00064 void PriorityMap::Set(PORT_PRIO_TYPE uXPrio_)
00065 {
00066     PORT_PRIO_TYPE uXPrioBit = PRIO_BIT(uXPrio_);
00067 #if PRIO_MAP_MULTI_LEVEL
00068     PORT_PRIO_TYPE uXWordIdx = PRIO_MAP_WORD_INDEX(uXPrio_);
00069
00070     m_auXPriorityMap[uXWordIdx] |= (1 << uXPrioBit);
00071     m_uXPriorityMapL2 |= (1 << uXWordIdx);
00072 #else
00073     m_uXPriorityMap |= (1 << uXPrioBit);
00074 #endif
00075 }
00076
00077 //---------------------------------------------------------------------------
00078 void PriorityMap::Clear(PORT_PRIO_TYPE uXPrio_)
00079 {
00080     PORT_PRIO_TYPE uXPrioBit = PRIO_BIT(uXPrio_);
00081 #if PRIO_MAP_MULTI_LEVEL
00082     PORT_PRIO_TYPE uXWordIdx = PRIO_MAP_WORD_INDEX(uXPrio_);
00083
00084     m_auXPriorityMap[uXWordIdx] &= ~(1 << uXPrioBit);
00085     if (!m_auXPriorityMap[uXWordIdx]) {
00086         m_uXPriorityMapL2 &= ~(1 << uXWordIdx);
00087     }
00088 #else
00089     m_uXPriorityMap &= ~(1 << uXPrioBit);
00090 #endif
00091 }
00092
00093 //---------------------------------------------------------------------------
00094 PORT_PRIO_TYPE PriorityMap::HighestPriority(void)
00095 {
00096 #if PRIO_MAP_MULTI_LEVEL
00097     PORT_PRIO_TYPE uXMapIdx = priority_from_bitmap(m_uXPriorityMapL2);
00098     if (!uXMapIdx) {
00099         return 0;
00100     }
00101     uXMapIdx--;
00102     PORT_PRIO_TYPE uXPrio = priority_from_bitmap(m_auXPriorityMap[uXMapIdx]);
00103     uXPrio += (uXMapIdx * PRIO_MAP_BITS);
00104 #else
00105     PORT_PRIO_TYPE uXPrio = priority_from_bitmap(m_uXPriorityMap);
00106 #endif
00107     return uXPrio;
00108 }
```

20.47 /home/moslevin/mark3-source/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"
```

20.47.1 Detailed Description

Code profiling utilities.

Definition in file profile.cpp.

20.48 profile.cpp

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

```
00002        _____            _____            _____            _____
00003   ___|      _|__    __|__       |__    __|__     |__    __|__     |__    _____
00004  |     \  /   |  ||        \          ||         |    ||   |/ /       ||___    |
00005  |      \/    |  ||         \          ||         \         ||         \       ||___   |
00006  |__/\__/|__|_||__|\__\   __||__|\__\   __||__|\__\   __||_____|
00007       |_____|           |_____|           |_____|           |_____|
00008
00009  --[Mark3 Realtime Platform]----------------------------------------------------
00010
00011  Copyright (c) 2012-2016 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          CS_ENTER();
00056          m_u32CurrentIteration = 0;
00057          m_u32InitialEpoch     = Profiler::GetEpoch();
00058          m_u16Initial          = Profiler::Read();
00059          CS_EXIT();
00060          m_bActive = 1;
00061      }
00062  }
00063
00064  //---------------------------------------------------------------------------
00065  void ProfileTimer::Stop()
00066  {
00067      if (m_bActive) {
00068          uint16_t u16Final;
00069          uint32_t u32Epoch;
00070          CS_ENTER();
00071          u16Final = Profiler::Read();
00072          u32Epoch = Profiler::GetEpoch();
00073          // Compute total for current iteration...
00074          m_u32CurrentIteration = ComputeCurrentTicks(u16Final,
00075          u32Epoch);
00076          m_u32Cumulative += m_u32CurrentIteration;
00077          m_u16Iterations++;
00078          CS_EXIT();
00079          m_bActive = 0;
00080      }
00081  }
00082
00083  //---------------------------------------------------------------------------
00084  uint32_t ProfileTimer::GetAverage()
00085  {
00086      if (m_u16Iterations) {
00087          return m_u32Cumulative / (uint32_t)m_u16Iterations;
00088      }
00089      return 0;
00090  }
00091
00092  //---------------------------------------------------------------------------
00093  uint32_t ProfileTimer::GetCurrent()
00094  {
00095      if (m_bActive) {
```

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

20.49 /home/moslevin/mark3-source/embedded/kernel/public/atomic.h File Reference

Basic Atomic Operations.

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

20.49.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.h.

20.50 atomic.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_      |__  __|__      |__  __|__      |__  _____
00004 |    \  /  |  | ||     \      ||       |    || |/ /     ||___   |
00005 |     \/   | ||      \      ||      \      ||      \      ||___    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007     |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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__
```

20.51 /home/moslevin/mark3-source/embedded/kernel/public/autoalloc.h File Reference

Automatic memory allocation for kernel objects.

```
#include <stdint.h>
#include <stdbool.h>
#include "mark3cfg.h"
```

20.51.1 Detailed Description

Automatic memory allocation for kernel objects.

Definition in file autoalloc.h.

20.52 autoalloc.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_  |__    |__    _|__  |__  __|__  |__  _____
00004 |     \  /  |  ||    \       ||      |    ||  |/ /      ||___  |
00005 |      \/   |  ||     \      ||       \   ||  |  \      ||__   |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========================================================================== */
00020 #ifndef __AUTO_ALLOC_H__
00021 #define __AUTO_ALLOC_H__
00022
00023 #include <stdint.h>
00024 #include <stdbool.h>
00025 #include "mark3cfg.h"
00026
00027 #if KERNEL_USE_AUTO_ALLOC
00028 // Forward declaration of kernel objects that can be auotmatically allocated.
00029
00030 #if KERNEL_USE_EVENTFLAG
00031 class EventFlag;
00032 #endif
00033
00034 #if KERNEL_USE_MAILBOX
```

```
00035 class Mailbox;
00036 #endif
00037
00038 #if KERNEL_USE_MESSAGE
00039 class Message;
00040 class MessageQueue;
00041 #endif
00042
00043 #if KERNEL_USE_MUTEX
00044 class Mutex;
00045 #endif
00046
00047 #if KERNEL_USE_NOTIFY
00048 class Notify;
00049 #endif
00050
00051 #if KERNEL_USE_SEMAPHORE
00052 class Semaphore;
00053 #endif
00054
00055 class Thread;
00056
00057 #if KERNEL_USE_TIMERS
00058 class Timer;
00059 #endif
00060
00061 class AutoAlloc
00062 {
00063 public:
00070     static void Init(void);
00071
00082     static void* Allocate(uint16_t u16Size_);
00083
00084 #if KERNEL_USE_SEMAPHORE
00085     static Semaphore* NewSemaphore(void);
00086 #endif
00087
00088 #if KERNEL_USE_MUTEX
00089     static Mutex* NewMutex(void);
00090 #endif
00091
00092 #if KERNEL_USE_EVENTFLAG
00093     static EventFlag* NewEventFlag(void);
00094 #endif
00095
00096 #if KERNEL_USE_MESSAGE
00097     static Message*      NewMessage(void);
00098     static MessageQueue* NewMessageQueue(void);
00099 #endif
00100
00101 #if KERNEL_USE_NOTIFY
00102     static Notify* NewNotify(void);
00103 #endif
00104
00105 #if KERNEL_USE_MAILBOX
00106     static Mailbox* NewMailbox(void);
00107 #endif
00108
00109     static Thread* NewThread(void);
00110
00111 #if KERNEL_USE_TIMERS
00112     static Timer* NewTimer(void);
00113 #endif
00114
00115 private:
00116     static uint8_t m_au8AutoHeap[AUTO_ALLOC_SIZE]; // Heap memory
00117     static K_ADDR  m_aHeapTop;                     // Top of the heap
00118 };
00119 #endif
00120
00121 #endif
```

20.53   /home/moslevin/mark3-source/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 primitives.

20.53.1 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.

20.54 blocking.h

```
00001 /*===========================================================================
00002        _____        _____        _____        _____
00003   ___|    _|__    __|_    |__    __|__    |__    __|__    _____
00004  |    \  /  |   ||      \       ||        ||  |/ /        ||___    |
00005  |     \/   |   ||       \      ||        ||  \           ||__     |
00006  |__/\__/|__||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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
00098     void BlockPriority(Thread* pclThread_);
00099
00111     void UnBlock(Thread* pclThread_);
00112
00117     ThreadList m_clBlockList;
00118 };
00119
00120 #endif
00121
00122 #endif
```

20.55 /home/moslevin/mark3-source/embedded/kernel/public/buffalogger.h   File   Reference

Super-efficient, super-secure logging routines.

```
#include <stdint.h>
```

20.55.1   Detailed Description

Super-efficient, super-secure logging routines.

Uses offline processing to ensure performance.

Definition in file buffalogger.h.

20.56   buffalogger.h

```
00001 /*=========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__    _|__  __|_    |__    |__    _____
00004 |    \  /   |  | |    \      ||        |        ||    ||/ /      ||___  |
00005 |     \/    |  | |      \     ||        ||        ||    ||___    ||___   |
00006 |__/\__/|__|_||__|\__\ __||__|\__\  __||__|\__\  __||_____|
00007     |_____|       |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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     do {                                                                   \
00030         const static volatile char     log_str[] __attribute__((section(".logger")))
    __attribute__((unused)) = str;        \
00031         const static volatile uint16_t line_id __attribute__((section(".logger"))) __attribute__((unused))
    = __LINE__; \
00032         const static volatile uint16_t file_id __attribute__((section(".logger"))) __attribute__((unused))
    = DBG_FILE; \
00033         const static volatile uint16_t sync __attribute__((section(".logger"))) __attribute__((unused))
    = 0xCAFE;    \
00034     } while (0);
```

20.57   /home/moslevin/mark3-source/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.

20.57.1   Detailed Description

Driver abstraction framework.

Driver abstraction framework for Mark3C.

20.57.2   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.

20.57.3   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.

20.57.4   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.

20.58 driver.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /  |  | ||    \      ||    \      ||  |/ /     ||___   |
00005 |     \/   |  | ||     \     ||     \     ||   \      ||___    |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||__||_____|
00007     |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------
00010
00011 Copyright (c) 2012-2016 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:
00124     void* operator new(size_t sz, void* pv) { return (Driver*)pv; };
00130     virtual void Init() = 0;
00131
00139     virtual uint8_t Open() = 0;
00140
00148     virtual uint8_t Close() = 0;
00149
00164     virtual uint16_t Read(uint16_t u16Bytes_, uint8_t* pu8Data_) = 0;
00165
00181     virtual uint16_t Write(uint16_t u16Bytes_, uint8_t* pu8Data_) = 0;
00182
00201     virtual uint16_t
00202     Control(uint16_t u16Event_, void* pvDataIn_, uint16_t u16SizeIn_, void* pvDataOut_, uint16_t
    u16SizeOut_)
00203         = 0;
00204
00213     void SetName(const char* pcName_) { m_pcPath = pcName_; }
00221     const char* GetPath() { return m_pcPath; }
00222 private:
00224     const char* m_pcPath;
00225 };
```

```
00226
00227 //---------------------------------------------------------------------
00232 class DriverList
00233 {
00234 public:
00242     static void Init();
00243
00252     static void Add(Driver* pclDriver_) { m_clDriverList.
     Add(pclDriver_); }
00261     static void Remove(Driver* pclDriver_) { m_clDriverList.
     Remove(pclDriver_); }
00270     static Driver* FindByPath(const char* m_pcPath);
00271
00272 private:
00274     static DoubleLinkList m_clDriverList;
00275 };
00276
00277 #endif // KERNEL_USE_DRIVER
00278
00279 #endif
```

20.59 /home/moslevin/mark3-source/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.

20.59.1 Detailed Description

Event Flag Blocking Object/IPC-Object definition.

Definition in file eventflag.h.

20.60 eventflag.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /    | ||     \    ||     |    || |/ /     ||___    |
00005 |     \/     | ||      \   ||     \    ||     \     ||__     |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  __||_____|
00007      |____|        |____|        |____|        |____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------
00010
00011 Copyright (c) 2012-2016 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:
00049     void* operator new(size_t sz, void* pv) { return (EventFlag*)pv; };
00050     ~EventFlag();
00051
00055     void Init()
00056     {
00057         m_u16SetMask = 0;
00058         m_clBlockList.Init();
00059     }
00060
00068     uint16_t Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_);
00069
00070 #if KERNEL_USE_TIMEOUTS
00071
00079     uint16_t Wait(uint16_t u16Mask_, EventFlagOperation_t eMode_, uint32_t
        u32TimeMS_);
00080
00088     void WakeMe(Thread* pclOwner_);
00089
00090 #endif
00091
00097     void Set(uint16_t u16Mask_);
00098
00103     void Clear(uint16_t u16Mask_);
00104
00109     uint16_t GetMask();
00110
00111 private:
00112 #if KERNEL_USE_TIMEOUTS
00113
00125     uint16_t Wait_i(uint16_t u16Mask_, EventFlagOperation_t eMode_, uint32_t
        u32TimeMS_);
00126 #else
00127
00137     uint16_t Wait_i(uint16_t u16Mask_, EventFlagOperation_t eMode_);
00138 #endif
00139
00140     uint16_t m_u16SetMask;
00141 };
00142
00143 #endif // KERNEL_USE_EVENTFLAG
00144 #endif //__EVENTFLAG_H__
```

20.61   /home/moslevin/mark3-source/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.

20.61.1   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.

20.62   kernel.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    __|_    |__    __|_    |__   |__    _____
00004 |    \  /   |   | ||      \      ||      \      ||   |/ /      ||___  |
00005 |     \/    | | ||       \       ||      \       ||     \       ||__    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007     |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 //---------------------------------------------------------------------------
00044 class Kernel
00045 {
00046 public:
00055     static void Init(void);
00056
00069     static void Start(void);
00070
00077     static bool IsStarted() { return m_bIsStarted; }
00085     static void SetPanic(PanicFunc_t pfPanic_) { m_pfPanic = pfPanic_; }
00090     static bool IsPanic() { return m_bIsPanic; }
00095     static void Panic(uint16_t u16Cause_);
00096
00097 #if KERNEL_USE_IDLE_FUNC
00098
00103     static void SetIdleFunc(IdleFunc_t pfIdle_) { m_pfIdle = pfIdle_; }
00108     static void IdleFunc(void)
00109     {
00110         if (m_pfIdle != 0) {
00111             m_pfIdle();
00112         }
00113     }
00114
00122     static Thread* GetIdleThread(void) { return (Thread*)&
    m_clIdle; }
00123 #endif
00124
00125 #if KERNEL_USE_THREAD_CALLOUTS
00126
00136     static void SetThreadCreateCallout(ThreadCreateCallout_t pfCreate_) {
    m_pfThreadCreateCallout = pfCreate_; }
00148     static void SetThreadExitCallout(ThreadExitCallout_t pfExit_) {
    m_pfThreadExitCallout = pfExit_; }
00159     static void SetThreadContextSwitchCallout(ThreadContextCallout_t
    pfContext_)
00160     {
00161         m_pfThreadContextCallout = pfContext_;
00162     }
00163
00172     static ThreadCreateCallout_t GetThreadCreateCallout(void) { return
    m_pfThreadCreateCallout; }
00181     static ThreadExitCallout_t GetThreadExitCallout(void) { return
    m_pfThreadExitCallout; }
00190     static ThreadContextCallout_t GetThreadContextSwitchCallout(void) { return
     m_pfThreadContextCallout; }
00191 #endif
00192
00193 #if KERNEL_USE_STACK_GUARD
00194     static void SetStackGuardThreshold(uint16_t u16Threshold_) { m_u16GuardThreshold = u16Threshold_; }
00195     static uint16_t                          GetStackGuardThreshold(void) { return m_u16GuardThreshold;
    }
00196 #endif
00197
```

```
00198 private:
00199     static bool        m_bIsStarted;
00200     static bool        m_bIsPanic;
00201     static PanicFunc_t m_pfPanic;
00202 #if KERNEL_USE_IDLE_FUNC
00203     static IdleFunc_t   m_pfIdle;
00204     static FakeThread_t m_clIdle;
00205 #endif
00206
00207 #if KERNEL_USE_THREAD_CALLOUTS
00208     static ThreadCreateCallout_t   m_pfThreadCreateCallout;
00209     static ThreadExitCallout_t     m_pfThreadExitCallout;
00210     static ThreadContextCallout_t  m_pfThreadContextCallout;
00211 #endif
00212
00213 #if KERNEL_USE_STACK_GUARD
00214     static uint16_t m_u16GuardThreshold;
00215 #endif
00216 };
00217
00218 #endif
```

20.63 /home/moslevin/mark3-source/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.

20.63.1 Detailed Description

Kernel aware simulation support.

Definition in file kernelaware.h.

20.63.2 Enumeration Type Documentation

20.63.2.1 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.

20.64 kernelaware.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__    _|__  __|_    |__   _____
00004 |    \  /    |  |    \    |      |    |    |    | /  /    |    |___    |
00005 |     \/     |  |    |     \    |      |    \    |     |    \     | |___     |
00006 |__/\__/|__|_|__|__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007      |____|        |____|        |____|        |____|
00008
00009 --[Mark3 Realtime Platform]------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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     KA_COMMAND_IDLE = 0,
00035     KA_COMMAND_PROFILE_INIT,
00036     KA_COMMAND_PROFILE_START,
00037     KA_COMMAND_PROFILE_STOP,
00038     KA_COMMAND_PROFILE_REPORT,
00039     KA_COMMAND_EXIT_SIMULATOR,
00040     KA_COMMAND_TRACE_0,
00041     KA_COMMAND_TRACE_1,
00042     KA_COMMAND_TRACE_2,
00043     KA_COMMAND_PRINT
00044 } KernelAwareCommand_t;
00045
00046 //--------------------------------------------------------------------------
00064 class KernelAware
00065 {
00066 public:
00067     //----------------------------------------------------------------------
00078     static void ProfileInit(const char* szStr_);
00079
00080     //----------------------------------------------------------------------
00088     static void ProfileStart(void);
00089
00090     //----------------------------------------------------------------------
00097     static void ProfileStop(void);
00098
00099     //----------------------------------------------------------------------
00107     static void ProfileReport(void);
00108
00109     //----------------------------------------------------------------------
00117     static void ExitSimulator(void);
00118
00119     //----------------------------------------------------------------------
00127     static void Print(const char* szStr_);
00128
```

```
00129      //---------------------------------------------------------------------
00139      static void Trace(uint16_t u16File_, uint16_t u16Line_);
00140
00141      //---------------------------------------------------------------------
00152      static void Trace(uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_);
00153
00154      //---------------------------------------------------------------------
00166      static void Trace(uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_);
00167
00168      //---------------------------------------------------------------------
00178      static bool IsSimulatorAware(void);
00179
00180 private:
00181      //---------------------------------------------------------------------
00194      static void
00195      Trace_i(uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_,
00195      KernelAwareCommand_t eCmd_);
00196 };
00197
00198 #endif
00199
00200 #endif
```

20.65 /home/moslevin/mark3-source/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"
```

Macros

- #define KERNEL_TRACE(x)

    Null Kernel Trace Macro.
- #define KERNEL_TRACE_1(x, arg1)

    Null Kernel Trace Macro.
- #define KERNEL_TRACE_2(x, arg1, arg2)

    Null Kernel Trace Macro.
- #define KERNEL_ASSERT(x)

    Null Kernel Assert Macro.

20.65.1 Detailed Description

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

Definition in file kerneldebug.h.

20.66 kerneldebug.h

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

```
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007      |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 && KERNEL_ENABLE_LOGGING)
00033
00034 //----------------------------------------------------------------------------
00035 #define KERNEL_TRACE(x)                                          \
00036     \
00037 {                                                                \
00038         EMIT_DBG_STRING(x);                                      \
00039         uint16_t au16Msg__[4];                                   \
00040         au16Msg__[0] = 0xACDC;                                   \
00041         au16Msg__[1] = DBG_FILE;                                 \
00042         au16Msg__[2] = __LINE__;                                 \
00043         au16Msg__[3] = TraceBuffer::Increment();                 \
00044         TraceBuffer::Write(au16Msg__, 4);                        \
00045     \
00046 };
00047
00048 //----------------------------------------------------------------------------
00049 #define KERNEL_TRACE_1(x, arg1)                                  \
00050     \
00051 {                                                                \
00052         EMIT_DBG_STRING(x);                                      \
00053         uint16_t au16Msg__[5];                                   \
00054         au16Msg__[0] = 0xACDC;                                   \
00055         au16Msg__[1] = DBG_FILE;                                 \
00056         au16Msg__[2] = __LINE__;                                 \
00057         au16Msg__[3] = TraceBuffer::Increment();                 \
00058         au16Msg__[4] = arg1;                                     \
00059         TraceBuffer::Write(au16Msg__, 5);                        \
00060     \
00061 }
00062
00063 //----------------------------------------------------------------------------
00064 #define KERNEL_TRACE_2(x, arg1, arg2)                            \
00065     \
00066 {                                                                \
00067         EMIT_DBG_STRING(x);                                      \
00068         uint16_t au16Msg__[6];                                   \
00069         au16Msg__[0] = 0xACDC;                                   \
00070         au16Msg__[1] = DBG_FILE;                                 \
00071         au16Msg__[2] = __LINE__;                                 \
00072         au16Msg__[3] = TraceBuffer::Increment();
```

```
                       \
00073           au16Msg__[4] = arg1;
                       \
00074           au16Msg__[5] = arg2;
                       \
00075           TraceBuffer::Write(au16Msg__, 6);
                       \
00076       \
00077 }
00078
00079 //----------------------------------------------------------------------
00080 #define KERNEL_ASSERT(x)
                       \
00081       \
00082 {
                   \
00083           if ((x) == false) {
                       \
00084               EMIT_DBG_STRING("ASSERT FAILED");
                       \
00085               uint16_t au16Msg__[4];
                       \
00086               au16Msg__[0] = 0xACDC;
                       \
00087               au16Msg__[1] = DBG_FILE;
                       \
00088               au16Msg__[2] = __LINE__;
                       \
00089               au16Msg__[3] = TraceBuffer::Increment();
                       \
00090               TraceBuffer::Write(au16Msg__, 4);
                       \
00091               Kernel::Panic(PANIC_ASSERT_FAILED);
                       \
00092           }
                       \
00093       \
00094 }
00095 #elif (KERNEL_USE_DEBUG && KERNEL_AWARE_SIMULATION && KERNEL_ENABLE_LOGGING)
00096
00097 //----------------------------------------------------------------------
00098 #define KERNEL_TRACE(x)
                       \
00099       \
00100 {
                   \
00101           EMIT_DBG_STRING(x);
                       \
00102           KernelAware::Trace(DBG_FILE, __LINE__);
                       \
00103       \
00104 };
00105
00106 //----------------------------------------------------------------------
00107 #define KERNEL_TRACE_1(x, arg1)
                       \
00108       \
00109 {
                   \
00110           EMIT_DBG_STRING(x);
                       \
00111           KernelAware::Trace(DBG_FILE, __LINE__, arg1);
                       \
00112       \
00113 }
00114
00115 //----------------------------------------------------------------------
00116 #define KERNEL_TRACE_2(x, arg1, arg2)
                       \
00117       \
00118 {
                   \
00119           EMIT_DBG_STRING(x);
                       \
00120           KernelAware::Trace(DBG_FILE, __LINE__, arg1, arg2);
                       \
00121       \
00122 }
00123
00124 //----------------------------------------------------------------------
00125 #define KERNEL_ASSERT(x)
                       \
00126       \
00127 {
                   \
00128           if ((x) == false) {
                       \
```

```
00129                EMIT_DBG_STRING("ASSERT FAILED");
                                                                 \
00130                KernelAware::Trace(DBG_FILE, __LINE__);
                                                                 \
00131                Kernel::Panic(PANIC_ASSERT_FAILED);
                                                                 \
00132        }
                                                                 \
00133    \
00134 }
00135
00136 #else
00137 //----------------------------------------------------------------------------
00138 // Note -- when kernel-debugging is disabled, we still have to define the
00139 // macros to ensure that the expressions compile (albeit, by elimination
00140 // during pre-processing).
00141 //----------------------------------------------------------------------------
00142 #define KERNEL_TRACE(x)
00143 //----------------------------------------------------------------------------
00144 #define KERNEL_TRACE_1(x, arg1)
00145 //----------------------------------------------------------------------------
00146 #define KERNEL_TRACE_2(x, arg1, arg2)
00147 //----------------------------------------------------------------------------
00148 #define KERNEL_ASSERT(x)
00149
00150 #endif // KERNEL_USE_DEBUG
00151
00152 //----------------------------------------------------------------------------
00153 #if (KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION && KERNEL_ENABLE_USER_LOGGING)
00154
00155 //----------------------------------------------------------------------------
00156 #define USER_TRACE(x)
                                                                 \
00157    \
00158 {
                                                                 \
00159        EMIT_DBG_STRING(x);
                                                                 \
00160        uint16_t au16Msg__[4];
                                                                 \
00161        au16Msg__[0] = 0xACDC;
                                                                 \
00162        au16Msg__[1] = DBG_FILE;
                                                                 \
00163        au16Msg__[2] = __LINE__;
                                                                 \
00164        au16Msg__[3] = TraceBuffer::Increment();
                                                                 \
00165        TraceBuffer::Write(au16Msg__, 4);
                                                                 \
00166    \
00167 };
00168
00169 //----------------------------------------------------------------------------
00170 #define USER_TRACE_1(x, arg1)
                                                                 \
00171    \
00172 {
                                                                 \
00173        EMIT_DBG_STRING(x);
                                                                 \
00174        uint16_t au16Msg__[5];
                                                                 \
00175        au16Msg__[0] = 0xACDC;
                                                                 \
00176        au16Msg__[1] = DBG_FILE;
                                                                 \
00177        au16Msg__[2] = __LINE__;
                                                                 \
00178        au16Msg__[3] = TraceBuffer::Increment();
                                                                 \
00179        au16Msg__[4] = arg1;
                                                                 \
00180        TraceBuffer::Write(au16Msg__, 5);
                                                                 \
00181    \
00182 }
00183
00184 //----------------------------------------------------------------------------
00185 #define USER_TRACE_2(x, arg1, arg2)
                                                                 \
00186    \
00187 {
                                                                 \
00188        EMIT_DBG_STRING(x);
                                                                 \
00189        uint16_t au16Msg__[6];
```

```
                        \
00190           au16Msg__[0] = 0xACDC;
                        \
00191           au16Msg__[1] = DBG_FILE;
                        \
00192           au16Msg__[2] = __LINE__;
                        \
00193           au16Msg__[3] = TraceBuffer::Increment();
                        \
00194           au16Msg__[4] = arg1;
                        \
00195           au16Msg__[5] = arg2;
                        \
00196           TraceBuffer::Write(au16Msg__, 6);
                        \
00197       \
00198 }
00199
00200 //---------------------------------------------------------------------------
00201 #define USER_ASSERT(x)
                        \
00202       \
00203 {
                        \
00204       if ((x) == false) {
                        \
00205           EMIT_DBG_STRING("ASSERT FAILED");
                        \
00206           uint16_t au16Msg__[4];
                        \
00207           au16Msg__[0] = 0xACDC;
                        \
00208           au16Msg__[1] = DBG_FILE;
                        \
00209           au16Msg__[2] = __LINE__;
                        \
00210           au16Msg__[3] = TraceBuffer::Increment();
                        \
00211           TraceBuffer::Write(au16Msg__, 4);
                        \
00212           Kernel::Panic(PANIC_ASSERT_FAILED);
                        \
00213       }
                        \
00214       \
00215 }
00216 #elif (KERNEL_USE_DEBUG && KERNEL_AWARE_SIMULATION && KERNEL_ENABLE_USER_LOGGING)
00217
00218 //---------------------------------------------------------------------------
00219 #define USER_TRACE(x)
                        \
00220       \
00221 {
                        \
00222       EMIT_DBG_STRING(x);
                        \
00223       KernelAware::Trace(DBG_FILE, __LINE__);
                        \
00224       \
00225 };
00226
00227 //---------------------------------------------------------------------------
00228 #define USER_TRACE_1(x, arg1)
                        \
00229       \
00230 {
                        \
00231       EMIT_DBG_STRING(x);
                        \
00232       KernelAware::Trace(DBG_FILE, __LINE__, arg1);
                        \
00233       \
00234 }
00235
00236 //---------------------------------------------------------------------------
00237 #define USER_TRACE_2(x, arg1, arg2)
                        \
00238       \
00239 {
                        \
00240       EMIT_DBG_STRING(x);
                        \
00241       KernelAware::Trace(DBG_FILE, __LINE__, arg1, arg2);
                        \
00242       \
00243 }
00244
```

```
00245 //---------------------------------------------------------------------------
00246 #define USER_ASSERT(x)                                                      \
00247     \
00248 {                                                                           \
00249         if ((x) == false) {                                                 \
00250             EMIT_DBG_STRING("ASSERT FAILED");                               \
00251             KernelAware::Trace(DBG_FILE, __LINE__);                         \
00252             Kernel::Panic(PANIC_ASSERT_FAILED);                             \
00253         }                                                                   \
00254     \
00255 }
00256
00257 #else
00258 //---------------------------------------------------------------------------
00259 // Note -- when kernel-debugging is disabled, we still have to define the
00260 // macros to ensure that the expressions compile (albeit, by elimination
00261 // during pre-processing).
00262 //---------------------------------------------------------------------------
00263 #define USER_TRACE(x)
00264 //---------------------------------------------------------------------------
00265 #define USER_TRACE_1(x, arg1)
00266 //---------------------------------------------------------------------------
00267 #define USER_TRACE_2(x, arg1, arg2)
00268 //---------------------------------------------------------------------------
00269 #define USER_ASSERT(x)
00270
00271 #endif // KERNEL_USE_DEBUG
00272
00273 #endif
```

20.67 /home/moslevin/mark3-source/embedded/kernel/public/kerneltypes.h File Reference

Basic data type primatives used throughout the OS.

```
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
```

Typedefs

- typedef void(∗ PanicFunc_t)(uint16_t u16PanicCode_)

    Function pointer type used to implement kernel-panic handlers.

- typedef void(∗ IdleFunc_t)(void)

    Function pointer type used to implement the idle function, where support for an idle function (as opposed to an idle thread) exists.

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

    Function pointer type used for thread entrypoint functions.

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.

- enum ThreadState_t

    Enumeration representing the different states a thread can exist in.

20.67.1 Detailed Description

Basic data type primatives used throughout the OS.

Definition in file kerneltypes.h.

20.67.2 Enumeration Type Documentation

20.67.2.1 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 50 of file kerneltypes.h.

20.68 kerneltypes.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__   __|_ |__     |__    _|__    |__    _|__   |__    _____
00004 |    \  /  |  | ||    \       ||     ||   |/ /     ||___   |
00005 |     \/   |  | ||     \      ||      \    ||    \      ||__    |
00006 |__/\__/|__|_||__|\__\   _||__|\__\  _||__|\__\  _||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00019 #include <stdint.h>
00020 #include <stdbool.h>
00021 #include <stddef.h>
00022
00023 #ifndef __KERNELTYPES_H__
00024 #define __KERNELTYPES_H__
00025
00026 //-----------------------------------------------------------------------
00030 typedef void (*PanicFunc_t)(uint16_t u16PanicCode_);
00031
00032 //-----------------------------------------------------------------------
00037 typedef void (*IdleFunc_t)(void);
00038
00039 //-----------------------------------------------------------------------
00043 typedef void (*ThreadEntry_t)(void* pvArg_);
00044
00045 //-----------------------------------------------------------------------
00050 typedef enum {
00051     EVENT_FLAG_ALL,
00052     EVENT_FLAG_ANY,
00053     EVENT_FLAG_ALL_CLEAR,
00054     EVENT_FLAG_ANY_CLEAR,
00055                                     //---
00056     EVENT_FLAG_MODES,
00057     EVENT_FLAG_PENDING_UNBLOCK
00058 } EventFlagOperation_t;
00059
00060 //-----------------------------------------------------------------------
00064 typedef enum {
00065     THREAD_STATE_EXIT = 0,
00066     THREAD_STATE_READY,
00067     THREAD_STATE_BLOCKED,
```

```
00068      THREAD_STATE_STOP,
00069      //--
00070      THREAD_STATES
00071 } ThreadState_t;
00072
00073 #endif
```

20.69   /home/moslevin/mark3-source/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

   Binary & Counting semaphores, based on BlockingObject base class.

20.69.1   Detailed Description

Semaphore Blocking Object class declarations.

Definition in file ksemaphore.h.

20.70   ksemaphore.h

```
00001 /*=========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__    _|__  __|_    |__  _____
00004 |    \  /   |    |    \     |    |    |      |    |/ /    |  |___|     |
00005 |     \/    |    |    |    \       |    |    \      |    |    \        |  |___|    |
00006 |__/\__/|__|_|_|__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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:
00040      void* operator new(size_t sz, void* pv) { return (Semaphore*)pv; };
00041      ~Semaphore();
00042
00064      void Init(uint16_t u16InitVal_, uint16_t u16MaxVal_);
00065
00080      bool Post();
00081
00089      void Pend();
```

```
00090
00102     uint16_t GetCount();
00103
00104 #if KERNEL_USE_TIMEOUTS
00105
00116     bool Pend(uint32_t u32WaitTimeMS_);
00117
00128     void WakeMe(Thread* pclChosenOne_);
00129 #endif
00130
00131 private:
00137     uint8_t WakeNext();
00138
00139 #if KERNEL_USE_TIMEOUTS
00140
00148     bool Pend_i(uint32_t u32WaitTimeMS_);
00149 #else
00150
00156     void Pend_i(void);
00157 #endif
00158
00159     uint16_t m_u16Value;
00160     uint16_t m_u16MaxValue;
00161 };
00162
00163 #endif // KERNEL_USE_SEMAPHORE
00164
00165 #endif
```

20.71 /home/moslevin/mark3-source/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.

20.71.1 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.

20.72   ll.h

```
00001 /*===========================================================================
00002      _____        _____       _____       _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|__|___|__  __|_    ____
00004 |    \  /   | ||    \     ||    |      ||  |/ /      ||___    |
00005 |     \/    | ||     \    ||    |      ||  |/ /      ||__    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------
00010
00011 Copyright (c) 2012-2016 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     LinkListNode* next;
00072     LinkListNode* prev;
00073
00074     LinkListNode() {}
00080     void ClearNode();
00081
00082 public:
00090     LinkListNode* GetNext(void) { return next; }
00098     LinkListNode* GetPrev(void) { return prev; }
00099     friend class LinkList;
00100     friend class DoubleLinkList;
00101     friend class CircularLinkList;
00102     friend class ThreadList;
00103 };
00104
00105 //---------------------------------------------------------------------------
00109 class LinkList
00110 {
00111 protected:
00112     LinkListNode* m_pstHead;
00113     LinkListNode* m_pstTail;
00114
00115 public:
00121     void Init()
00122     {
00123         m_pstHead = NULL;
00124         m_pstTail = NULL;
00125     }
00126
00134     LinkListNode* GetHead() { return m_pstHead; }
00142     LinkListNode* GetTail() { return m_pstTail; }
00143 };
00144
00145 //---------------------------------------------------------------------------
00149 class DoubleLinkList : public LinkList
00150 {
00151 public:
00152     void* operator new(size_t sz, void* pv) { return (DoubleLinkList*)pv; };
00158     DoubleLinkList()
00159     {
00160         m_pstHead = NULL;
00161         m_pstTail = NULL;
00162     }
00163
00171     void Add(LinkListNode* node_);
00172
00180     void Remove(LinkListNode* node_);
00181 };
00182
00183 //---------------------------------------------------------------------------
00187 class CircularLinkList : public LinkList
00188 {
```

```
00189 public:
00190     void* operator new(size_t sz, void* pv) { return (CircularLinkList*)pv; };
00191     CircularLinkList()
00192     {
00193         m_pstHead = NULL;
00194         m_pstTail = NULL;
00195     }
00196
00204     void Add(LinkListNode* node_);
00205
00213     void Remove(LinkListNode* node_);
00214
00221     void PivotForward();
00222
00229     void PivotBackward();
00230
00240     void InsertNodeBefore(LinkListNode* node_,
      LinkListNode* insert_);
00241 };
00242
00243 #endif
```

20.73   /home/moslevin/mark3-source/embedded/kernel/public/mailbox.h File Reference

Mailbox + Envelope IPC Mechanism.

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

Classes

- class Mailbox

  The Mailbox class implements an IPC mechnism based on envelopes containing data of a fixed size (configured at initialization) that reside within a buffer of memory provided by the user.

20.73.1   Detailed Description

Mailbox + Envelope IPC Mechanism.

Definition in file mailbox.h.

20.74   mailbox.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    __|_    |__    |__    _|__  _____
00004 |    \  /    |  |    \    |    |      ||    |/ /      ||___  |
00005 |     \/     |  ||     \    ||      \    ||    \      ||___    |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007     |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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
00035 class Mailbox
00036 {
```

```
00037 public:
00038     void* operator new(size_t sz, void* pv) { return (Mailbox*)pv; };
00039     ~Mailbox();
00040
00051     void Init(void* pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_);
00052
00053 #if KERNEL_USE_AUTO_ALLOC
00054
00067     static Mailbox* Init(uint16_t u16BufferSize_, uint16_t u16ElementSize_);
00068
00069 #endif
00070
00084     bool Send(void* pvData_);
00085
00099     bool SendTail(void* pvData_);
00100
00101 #if KERNEL_USE_TIMEOUTS
00102
00116     bool Send(void* pvData_, uint32_t u32TimeoutMS_);
00117
00132     bool SendTail(void* pvData_, uint32_t u32TimeoutMS_);
00133 #endif
00134
00144     void Receive(void* pvData_);
00145
00155     void ReceiveTail(void* pvData_);
00156
00157 #if KERNEL_USE_TIMEOUTS
00158
00170     bool Receive(void* pvData_, uint32_t u32TimeoutMS_);
00171
00184     bool ReceiveTail(void* pvData_, uint32_t u32TimeoutMS_);
00185 #endif
00186
00187     uint16_t GetFreeSlots(void)
00188     {
00189         uint16_t rc;
00190         CS_ENTER();
00191         rc = m_u16Free;
00192         CS_EXIT();
00193         return rc;
00194     }
00195
00196     bool IsFull(void) { return (GetFreeSlots() == 0); }
00197     bool IsEmpty(void) { return (GetFreeSlots() == m_u16Count); }
00198 private:
00207     void* GetHeadPointer(void)
00208     {
00209         K_ADDR uAddr = (K_ADDR)m_pvBuffer;
00210         uAddr += (K_ADDR)(m_u16ElementSize) * (K_ADDR)(
00211     m_u16Head);
00211         return (void*)uAddr;
00212     }
00213
00222     void* GetTailPointer(void)
00223     {
00224         K_ADDR uAddr = (K_ADDR)m_pvBuffer;
00225         uAddr += (K_ADDR)(m_u16ElementSize) * (K_ADDR)(
00226     m_u16Tail);
00226         return (void*)uAddr;
00227     }
00228
00238     void CopyData(const void* src_, const void* dst_, uint16_t len_)
00239     {
00240         uint8_t* u8Src = (uint8_t*)src_;
00241         uint8_t* u8Dst = (uint8_t*)dst_;
00242         while (len_--) {
00243             *u8Dst++ = *u8Src++;
00244         }
00245     }
00246
00252     void MoveTailForward(void)
00253     {
00254         m_u16Tail++;
00255         if (m_u16Tail == m_u16Count) {
00256             m_u16Tail = 0;
00257         }
00258     }
00259
00265     void MoveHeadForward(void)
00266     {
00267         m_u16Head++;
00268         if (m_u16Head == m_u16Count) {
00269             m_u16Head = 0;
00270         }
00271     }
00272
```

```
00278     void MoveTailBackward(void)
00279     {
00280         if (m_u16Tail == 0) {
00281             m_u16Tail = m_u16Count;
00282         }
00283         m_u16Tail--;
00284     }
00285
00291     void MoveHeadBackward(void)
00292     {
00293         if (m_u16Head == 0) {
00294             m_u16Head = m_u16Count;
00295         }
00296         m_u16Head--;
00297     }
00298
00299 #if KERNEL_USE_TIMEOUTS
00300
00310     bool Send_i(const void* pvData_, bool bTail_, uint32_t u32WaitTimeMS_);
00311 #else
00312
00321     bool Send_i(const void* pvData_, bool bTail_);
00322 #endif
00323
00324 #if KERNEL_USE_TIMEOUTS
00325
00335     bool Receive_i(const void* pvData_, bool bTail_, uint32_t u32WaitTimeMS_);
00336 #else
00337
00345     void Receive_i(const void* pvData_, bool bTail_);
00346 #endif
00347
00348     uint16_t m_u16Head;
00349     uint16_t m_u16Tail;
00350
00351     uint16_t        m_u16Count;
00352     volatile uint16_t m_u16Free;
00353
00354     uint16_t     m_u16ElementSize;
00355     const void* m_pvBuffer;
00356
00357     Semaphore m_clRecvSem;
00358
00359 #if KERNEL_USE_TIMEOUTS
00360     Semaphore m_clSendSem;
00361 #endif
00362 };
00363
00364 #endif
00365
00366 #endif
```

20.75 /home/moslevin/mark3-source/embedded/kernel/public/manual.h File Reference

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

20.75.1 Detailed Description

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

Definition in file manual.h.

20.76 manual.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    _|__  __|_    _|__  __|_    _|__  _____
00004 |    \  /    |  | |    \       | |    |       | |  |/ /     ||___   |
00005 |     \/     |  | |     \      | |     \      | |   \       ||___    |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
```

20.77 /home/moslevin/mark3-source/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"
#include "autoalloc.h"
#include "priomap.h"
```

20.77.1 Detailed Description

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

Definition in file mark3.h.

20.78 mark3.h

```
00001 /*============================================================================
00002        _____        _____        _____        _____
00003  ___|    _|__    __|_    |__    __|_    |__    __|    __|    __|    ____
00004 |    \  /    |   |  | |       \       | |     || |/ /     ||___    |
00005 |     \/     |   | ||       \       ||      \       ||      ||___    |
00006 |__/\__/|__|_||__|\__\   _||__|\__\   _||__|\__\    _||_____|
00007       |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]----------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 #include "autoalloc.h"
00050 #include "priomap.h"
00051
00052 #endif
```

20.79 /home/moslevin/mark3-source/embedded/kernel/public/mark3cfg.h File Reference

Mark3 Kernel Configuration.

```
#include "portcfg.h"
```

Macros

- #define KERNEL_NUM_PRIORITIES (8)

    Define the number of thread priorities that the kernel's scheduler will support.

- #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_TIMERS_MINIMUM_DELAY_US (25)

    When using tickless timers, it is useful to define a minimum sleep value.

- #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_ENABLE_LOGGING (0)

> *Set this to 1 to enable very chatty kernel logging.*

- #define KERNEL_ENABLE_USER_LOGGING (1)

> *This enables a set of logging macros similar to the kernel-logging macros; however, these can be enabled or disabled independently.*

- #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.*

- #define KERNEL_USE_AUTO_ALLOC (0)

> *This feature enables an additional set of APIs that allow for objects to be created on-the-fly out of a special heap, without having to explicitly allocate them (from stack, heap, or static memory).*

- #define KERNEL_USE_THREAD_CALLOUTS (1)

> *This feature provides additional kernel APIs to register callout functions that are activated when threads are created or exited.*

- #define KERNEL_USE_STACK_GUARD (0)

> *This feature, when enabled, tells the kernel to check whether any Thread's stack has been exhausted (or slack falls below a certain safety threshold) before executing each context switch.*

20.79.1   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.

20.79.2 Macro Definition Documentation

20.79.2.1 #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 174 of file mark3cfg.h.

20.79.2.2 #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 277 of file mark3cfg.h.

20.79.2.3 #define KERNEL_ENABLE_LOGGING (0)

Set this to 1 to enable very chatty kernel logging.

Since most important things in the kernel emit logs, a large log-buffer and fast output are required in order to keep up. This is a pretty advanced power-user type feature, so it's disabled by default.

Definition at line 242 of file mark3cfg.h.

20.79.2.4 #define KERNEL_ENABLE_USER_LOGGING (1)

This enables a set of logging macros similar to the kernel-logging macros; however, these can be enabled or disabled independently.

This allows for user-code to benefit from the built-in kernel logging macros without having to account for the super-high-volume of logs generated by kernel code.1 to enable logging outside of kernel code

Definition at line 251 of file mark3cfg.h.

20.79.2.5 #define KERNEL_NUM_PRIORITIES (8)

Define the number of thread priorities that the kernel's scheduler will support.

The number of thread priorities is limited only by the memory of the host CPU, as a ThreadList object is statically-allocated for each thread priority.

In practice, systems rarely need more than 32 priority levels, with the most complex having the capacity for 256.

Definition at line 41 of file mark3cfg.h.

20.79.2.6 #define KERNEL_TIMERS_MINIMUM_DELAY_US (25)

When using tickless timers, it is useful to define a minimum sleep value.

In the event that a delay/sleep/timeout value lower than this is provided to a timer-based API, the minimum value will be substituted.

Definition at line 86 of file mark3cfg.h.

20.79.2.7 #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 77 of file mark3cfg.h.

20.79.2.8 #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 261 of file mark3cfg.h.

20.79.2.9 #define KERNEL_USE_AUTO_ALLOC (0)

This feature enables an additional set of APIs that allow for objects to be created on-the-fly out of a special heap, without having to explicitly allocate them (from stack, heap, or static memory).

Note that auto-alloc memory cannot be reclaimed.

Definition at line 298 of file mark3cfg.h.

20.79.2.10 #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 221 of file mark3cfg.h.

20.79.2.11 #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 153 of file mark3cfg.h.

20.79.2.12 #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 287 of file mark3cfg.h.

20.79.2.13   #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 187 of file mark3cfg.h.

20.79.2.14   #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 161 of file mark3cfg.h.

20.79.2.15   #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 227 of file mark3cfg.h.

20.79.2.16   #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 116 of file mark3cfg.h.

20.79.2.17   #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 139 of file mark3cfg.h.

20.79.2.18   #define KERNEL_USE_STACK_GUARD (0)

This feature, when enabled, tells the kernel to check whether any Thread's stack has been exhausted (or slack falls below a certain safety threshold) before executing each context switch.

Enabling this is the most effective means to guard against stack corruption and stack overflow in the kernel, at the cost of increased context switch latency.include CPU/Port specific configuration options

Definition at line 320 of file mark3cfg.h.

20.79.2.19   #define KERNEL_USE_THREAD_CALLOUTS (1)

This feature provides additional kernel APIs to register callout functions that are activated when threads are created or exited.

This is useful for implementing low-level instrumentation based on information held in the threads.

Definition at line 310 of file mark3cfg.h.

20.79.2.20   #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 213 of file mark3cfg.h.

20.79.2.21   #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 101 of file mark3cfg.h.

20.79.2.22   #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 56 of file mark3cfg.h.

20.79.2.23   #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 269 of file mark3cfg.h.

20.79.2.24   #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 125 of file mark3cfg.h.

20.80 mark3cfg.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__    _|__   |__    _____
00004 |    \  /   |  | |    \    |   ||    |     ||  |/ /       ||___   |
00005 |     \/    |  | |     \    |   ||    |     ||  |   \     ||__    |
00006 |__/\__/|__|_||__|\__\   __||__|\__\   __||__|\__\   __||_____|
00007      |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]---------------------------------------------
00010
00011 Copyright (c) 2012-2016 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_NUM_PRIORITIES (8)
00042
00043 #if KERNEL_NUM_PRIORITIES > 1024
00044 #error "Mark3 supports a maximum of 1024 priorities"
00045 #endif
00046
00056 #define KERNEL_USE_TIMERS (1)
00057
00076 #if KERNEL_USE_TIMERS
00077 #define KERNEL_TIMERS_TICKLESS (1)
00078 #endif
00079
00080 #if KERNEL_TIMERS_TICKLESS
00081
00086 #define KERNEL_TIMERS_MINIMUM_DELAY_US (25)
00087 #endif
00088
00100 #if KERNEL_USE_TIMERS
00101 #define KERNEL_USE_TIMEOUTS (1)
00102 #else
00103 #define KERNEL_USE_TIMEOUTS (0)
00104 #endif
00105
00115 #if KERNEL_USE_TIMERS
00116 #define KERNEL_USE_QUANTUM (1)
00117 #else
00118 #define KERNEL_USE_QUANTUM (0)
00119 #endif
00120
00125 #define THREAD_QUANTUM_DEFAULT (4)
00126
00131 #define KERNEL_USE_NOTIFY (1)
00132
00139 #define KERNEL_USE_SEMAPHORE (1)
00140
00146 #define KERNEL_USE_MUTEX (1)
00147
00153 #define KERNEL_USE_EVENTFLAG (1)
00154
00160 #if KERNEL_USE_SEMAPHORE
00161 #define KERNEL_USE_MESSAGE (1)
00162 #else
00163 #define KERNEL_USE_MESSAGE (0)
00164 #endif
00165
00173 #if KERNEL_USE_MESSAGE
00174 #define GLOBAL_MESSAGE_POOL_SIZE (8)
00175 #endif
00176
00186 #if KERNEL_USE_SEMAPHORE
00187 #define KERNEL_USE_MAILBOX (1)
00188 #else
00189 #define KERNEL_USE_MAILBOX (0)
00190 #endif
00191
00196 #if KERNEL_USE_TIMERS && KERNEL_USE_SEMAPHORE
00197 #define KERNEL_USE_SLEEP (1)
00198 #else
00199 #define KERNEL_USE_SLEEP (0)
00200 #endif
00201
00206 #define KERNEL_USE_DRIVER (1)
00207
00213 #define KERNEL_USE_THREADNAME (0)
00214
00221 #define KERNEL_USE_DYNAMIC_THREADS (1)
00222
00227 #define KERNEL_USE_PROFILER (1)
```

```
00228
00233 #define KERNEL_USE_DEBUG (1)
00234
00235 #if KERNEL_USE_DEBUG
00236
00242 #define KERNEL_ENABLE_LOGGING (0)
00243
00251 #define KERNEL_ENABLE_USER_LOGGING (1)
00252 #else
00253 #define KERNEL_ENABLE_LOGGING (0)
00254 #define KERNEL_ENABLE_USER_LOGGING (0)
00255 #endif
00256
00261 #define KERNEL_USE_ATOMIC (0)
00262
00269 #define SAFE_UNLINK (0)
00270
00277 #define KERNEL_AWARE_SIMULATION (1)
00278
00286 #if !defined(ARM)
00287 #define KERNEL_USE_IDLE_FUNC (1) // Supported everywhere but ARM
00288 #else
00289 #define KERNEL_USE_IDLE_FUNC (0) // Not currently supported on ARM
00290 #endif
00291
00298 #define KERNEL_USE_AUTO_ALLOC (0)
00299
00300 #if KERNEL_USE_AUTO_ALLOC
00301 #define AUTO_ALLOC_SIZE (512)
00302 #endif
00303
00310 #define KERNEL_USE_THREAD_CALLOUTS (1)
00311
00320 #define KERNEL_USE_STACK_GUARD (0)
00321
00322 #if KERNEL_USE_STACK_GUARD
00323 #define KERNEL_STACK_GUARD_DEFAULT (32) // words
00324 #endif
00325
00326 #include "portcfg.h"
00327
00328 #endif
```

20.81 /home/moslevin/mark3-source/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 MessagePool

  Implements a list of message objects.

- 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.

20.81.1 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.

20.81.2 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.

20.82 message.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__    |_    _|__    |_    _|__    |_    _____
00004 |    \  /    |  |    |    \       ||       ||  |/ /        ||___
00005 |     \/     |  ||        \       ||       \       ||       ||___   |
00006 |__/\__/|__|_||__|\__\   _||__|\__\   _||__|\__\   _||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------
00010
00011 Copyright (c) 2012-2016 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:
00102     void* operator new(size_t sz, void* pv) { return (Message*)pv; };
00108     void Init()
00109     {
00110         ClearNode();
00111         m_pvData  = NULL;
00112         m_u16Code = 0;
00113     }
00114
00122     void SetData(void* pvData_) { m_pvData = pvData_; }
00130     void* GetData() { return m_pvData; }
00138     void SetCode(uint16_t u16Code_) { m_u16Code = u16Code_; }
00146     uint16_t GetCode() { return m_u16Code; }
00147 private:
00149     void* m_pvData;
00150
00152     uint16_t m_u16Code;
00153 };
00154
00155 //---------------------------------------------------------------------------
00159 class MessagePool
00160 {
00161 public:
00167     void Init();
00168
00178     void Push(Message* pclMessage_);
00179
00188     Message* Pop();
00189
00197     Message* GetHead();
00198
00199 private:
00201     DoubleLinkList m_clList;
00202 };
00203
00204 //---------------------------------------------------------------------------
00208 class GlobalMessagePool
00209 {
00210 public:
00216     static void Init();
00217
00227     static void Push(Message* pclMessage_);
00228
00237     static Message* Pop();
00238
00246     static Message* GetHead();
00247
00255     static MessagePool* GetPool();
00256
00257 private:
00259     static Message m_aclMessagePool[
00    GLOBAL_MESSAGE_POOL_SIZE];
00260
00261     static MessagePool m_clPool;
00262 };
00263
00264 //---------------------------------------------------------------------------
00269 class MessageQueue
00270 {
00271 public:
00272     void* operator new(size_t sz, void* pv) { return (MessageQueue*)pv; };
00278     void Init();
00279
00288     Message* Receive();
00289
00290 #if KERNEL_USE_TIMEOUTS
00291
00305     Message* Receive(uint32_t u32TimeWaitMS_);
00306 #endif
00307
00316     void Send(Message* pclSrc_);
00317
00325     uint16_t GetCount();
00326
00327 private:
00328 #if KERNEL_USE_TIMEOUTS
```

```
00329
00338     Message* Receive_i(uint32_t u32TimeWaitMS_);
00339 #else
00340
00347     Message* Receive_i(void);
00348 #endif
00349
00351     Semaphore m_clSemaphore;
00352
00354     DoubleLinkList m_clLinkList;
00355 };
00356
00357 #endif // KERNEL_USE_MESSAGE
00358
00359 #endif
```

20.83 /home/moslevin/mark3-source/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.

20.83.1 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.

20.83.2 Initializing

Initializing a mutex object by calling:

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

20.83.3 Resource protection example

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

Definition in file mutex.h.

20.84 mutex.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_ \      |__ __|__    |__  __| _     |__ _____
00004 |    \  /  | | |    \       | |     |      | |  |/ /     | |___   |
00005 |     \/   | | ||    \      | |     \      | |  |  \      | |__    |
00006 |__/\__/|__| |__|\__\  __| |__|\__\  __| |__|\__\  __| |_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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:
00071     void* operator new(size_t sz, void* pv) { return (Mutex*)pv; };
00072     ~Mutex();
00073
00080     void Init();
00081
00099     void Claim();
00100
00101 #if KERNEL_USE_TIMEOUTS
00102
00113     bool Claim(uint32_t u32WaitTimeMS_);
00114
00127     void WakeMe(Thread* pclOwner_);
00128
00129 #endif
00130
00151     void Release();
00152
00153 private:
00159     uint8_t WakeNext();
00160
00161 #if KERNEL_USE_TIMEOUTS
00162
00170     bool Claim_i(uint32_t u32WaitTimeMS_);
00171 #else
00172
00178     void Claim_i(void);
00179 #endif
00180
00181     uint8_t m_u8Recurse;
00182     bool    m_bReady;
00183     uint8_t m_u8MaxPri;
00184     Thread* m_pclOwner;
00185 };
00186
00187 #endif // KERNEL_USE_MUTEX
00188
00189 #endif //__MUTEX_H_
```

20.85 /home/moslevin/mark3-source/embedded/kernel/public/notify.h File Reference

Lightweight thread notification - blocking object.

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

Classes

- class Notify

    The Notify class is a blocking object type, that allows one or more threads to wait for an event to occur before resuming operation.

20.85.1 Detailed Description

Lightweight thread notification - blocking object.

Definition in file notify.h.

20.86 notify.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__    _|__   |__    _____
00004 |    \  /    | ||    \      | ||       | ||  |/ /      ||___    |
00005 |     \/     | ||     \      | ||       | ||  |< \      ||__    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007     |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00021 #ifndef __NOTIFY_H__
00022 #define __NOTIFY_H__
00023
00024 #include "mark3cfg.h"
00025 #include "blocking.h"
00026
00027 #if KERNEL_USE_NOTIFY
00028
00033 class Notify : public BlockingObject
00034 {
00035 public:
00036     void* operator new(size_t sz, void* pv) { return (Notify*)pv; };
00037     ~Notify();
00038
00044     void Init(void);
00045
00055     void Signal(void);
00056
00066     void Wait(bool* pbFlag_);
00067
00068 #if KERNEL_USE_TIMEOUTS
00069
00081     bool Wait(uint32_t u32WaitTimeMS_, bool* pbFlag_);
00082 #endif
00083
00093     void WakeMe(Thread* pclChosenOne_);
00094 };
00095
00096 #endif
00097
00098 #endif
```

20.87 /home/moslevin/mark3-source/embedded/kernel/public/paniccodes.h File Reference

Defines the reason codes thrown when a kernel panic occurs.

20.87.1 Detailed Description

Defines the reason codes thrown when a kernel panic occurs.

Definition in file paniccodes.h.

20.88   paniccodes.h

```
00001 /*=============================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__    |     \      |    _|__     |     \      |__    __|     |__    _____
00004 |    \  /    |   |      \     |    |        |     |      |  |  /  /        |   |__    |
00005 |     \/     |   |       \    |    |        |     |      |  |  \  \        |   |___   |
00006 |__/\__/|__|_||__|\__\    __||__|\__\    __||__|\__\    __|     |
00007      |____|        |____|        |____|        |____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 #define PANIC_AUTO_HEAP_EXHAUSTED (4)
00027 #define PANIC_POWERMAN_EXHAUSTED (5)
00028 #define PANIC_NO_READY_THREADS (6)
00029 #define PANIC_RUNNING_THREAD_DESCOPED (7)
00030 #define PANIC_ACTIVE_SEMAPHORE_DESCOPED (8)
00031 #define PANIC_ACTIVE_MUTEX_DESCOPED (9)
00032 #define PANIC_ACTIVE_EVENTFLAG_DESCOPED (10)
00033 #define PANIC_ACTIVE_NOTIFY_DESCOPED (11)
00034 #define PANIC_ACTIVE_MAILBOX_DESCOPED (12)
00035 #define PANIC_ACTIVE_TIMER_DESCOPED (13)
00036
00037 #endif // __PANIC_CODES_H
```

20.89   /home/moslevin/mark3-source/embedded/kernel/public/priomap.h File Reference

Priority map data structure.

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

Classes

- class PriorityMap

    The PriorityMap class.

20.89.1   Detailed Description

Priority map data structure.

Definition in file priomap.h.

20.90   priomap.h

```
00001 /*=============================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__    |     \      |    _|__     |     \      |__    __|     |__    _____
00004 |    \  /    |   |      \     |    |        |     |      |  |  /  /        |   |__    |
00005 |     \/     |   |       \    |    |        |     |      |  |  \  \        |   |___   |
00006 |__/\__/|__|_||__|\__\    __||__|\__\    __||__|\__\    __|     |
00007      |____|        |____|        |____|        |____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
```

```
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =====================================================================*/
00019 #ifndef __PRIOMAP_H__
00020 #define __PRIOMAP_H__
00021
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 //---------------------------------------------------------------------
00026 // Define the type used to store the priority map based on the word size of
00027 // the underlying host architecture.
00028 #if !defined(PORT_PRIO_MAP_WORD_SIZE)
00029 #error "undefined PORT_PRIO_MAP_WORD_SIZE"
00030 #endif
00031
00032 #define PRIO_MAP_WORD_TYPE K_WORD
00033
00034 // Size of the map index type in bits
00035 #define PRIO_MAP_BITS (8 * PORT_PRIO_MAP_WORD_SIZE)
00036
00037 // # of bits in an integer used to represent the number of bits in the map.
00038 // Used for bitshifting the bit index away from the map index.
00039 // i.e. 3 == 8 bits, 4 == 16 bits, 5 == 32 bits, etc...
00040 #define PRIO_MAP_WORD_SHIFT (2 + PORT_PRIO_MAP_WORD_SIZE)
00041
00042 // Bitmask used to separate out the priorities first-level bitmap from its
00043 // second-level map index for a given priority
00044 #define PRIO_MAP_BIT_MASK ((1 << PRIO_MAP_WORD_SHIFT) - 1)
00045
00046 // Get the priority bit for a given thread
00047 #define PRIO_BIT(x) ((x)&PRIO_MAP_BIT_MASK)
00048
00049 // Macro used to get the map index for a given priroity
00050 #define PRIO_MAP_WORD_INDEX(prio) ((prio) >> PRIO_MAP_WORD_SHIFT)
00051
00052 // Required size of the bitmap array in words
00053 #define PRIO_MAP_NUM_WORDS ((KERNEL_NUM_PRIORITIES + (PRIO_MAP_BITS - 1)) / (PRIO_MAP_BITS))
00054
00055 //---------------------------------------------------------------------
00056 #if (PRIO_MAP_NUM_WORDS == 1)
00057 // If there is only 1 word required to store the priority information, we don't
00058 // need an array, or a secondary bitmap.
00059 #define PRIO_MAP_MULTI_LEVEL (0)
00060 #else
00061 // An array of bitmaps are required, and a secondary index is required to
00062 // efficiently track which priority levels are active.
00063 #define PRIO_MAP_MULTI_LEVEL (1)
00064 #endif
00065
00066 //---------------------------------------------------------------------
00070 class PriorityMap
00071 {
00072 public:
00078     PriorityMap();
00079
00085     void Set(PORT_PRIO_TYPE uXPrio_);
00086
00092     void Clear(PORT_PRIO_TYPE uXPrio_);
00093
00102     PORT_PRIO_TYPE HighestPriority(void);
00103
00104 private:
00105 #if PRIO_MAP_MULTI_LEVEL
00106     PRIO_MAP_WORD_TYPE m_auXPriorityMap[PRIO_MAP_NUM_WORDS];
00107     PRIO_MAP_WORD_TYPE m_uXPriorityMapL2;
00108 #else
00109     PRIO_MAP_WORD_TYPE m_uXPriorityMap;
00110 #endif
00111 };
00112
00113 #endif
```

20.91 /home/moslevin/mark3-source/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.

20.91.1 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.

20.92   profile.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /    | |    \     |    |     |    | |  / /     |    |__ |
00005 |     \/     | |     \    |    |      |    | |     \     |    ___|
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007     |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========================================================================== */
00052 #ifndef __PROFILE_H__
00053 #define __PROFILE_H__
00054
00055 #include "kerneltypes.h"
00056 #include "mark3cfg.h"
00057 #include "ll.h"
00058
00059 #if KERNEL_USE_PROFILER
00060
00069 class ProfileTimer
00070 {
00071 public:
00078     void Init();
00079
00086     void Start();
00087
00094     void Stop();
00095
```

```
00103     uint32_t GetAverage();
00104
00113     uint32_t GetCurrent();
00114
00115 private:
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
```

20.93 /home/moslevin/mark3-source/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.

20.93.1 Detailed Description

Thread Quantum declarations for Round-Robin Scheduling.

Definition in file quantum.h.

20.94 quantum.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_   |__    __|_   |__    __|_   |____  _____
00004 |    \  /  |  |  |  |  \      |  |  |      ||  |/ /      ||___   |
00005 |     \/   |  |  |  |   \      |  |  |      ||  |  \      ||___   |
00006 |__/\__/|__|__||__|\__\   _||__|\__\   _||__|\__\   _||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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; }
00083     static void ClearInTimer(void) { m_bInTimer = false; }
00084 private:
00096     static void SetTimer(Thread* pclThread_);
00097
00098     static Timer m_clQuantumTimer;
00099     static bool  m_bActive;
00100     static bool  m_bInTimer;
00101 };
00102
00103 #endif // KERNEL_USE_QUANTUM
00104
00105 #endif
```

20.95 /home/moslevin/mark3-source/embedded/kernel/public/scheduler.h File Reference

Thread scheduler function declarations.

```
#include "kerneltypes.h"
#include "thread.h"
#include "threadport.h"
#include "priomap.h"
```

Classes

- class Scheduler

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

Variables

- volatile Thread ∗ g_pclNext

    Pointer to the currently-chosen next-running thread.

- Thread ∗ g_pclCurrent

    Pointer to the currently-running thread.

20.95.1 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.

20.96 scheduler.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__    __|_    |__    __|_    |__    __|_    |__    _____
00004 |    \  /   |  | |   \      | |      |    | |  | / /     | |___    |
00005 |     \/    |  | |    \     | |    \     | |  | \ \      | |___    |
00006 |__/\__/|__|_| |__|\__\   _| |__|\__\   _| |__|\__\   _| |____|
00007      |____|        |____|        |____|        |____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 #include "priomap.h"
00053
00054 extern volatile Thread* g_pclNext;
00055 extern Thread*          g_pclCurrent;
00056
00057 //--------------------------------------------------------------------------
00062 class Scheduler
00063 {
00064 public:
00070     static void Init();
00071
00079     static void Schedule();
00080
00088     static void Add(Thread* pclThread_);
00089
00098     static void Remove(Thread* pclThread_);
00099
00112     static bool SetScheduler(bool bEnable_);
00113
00121     static Thread* GetCurrentThread() { return
00     g_pclCurrent; }
00130     static volatile Thread* GetNextThread() { return
00     g_pclNext; }
00141     static ThreadList* GetThreadList(PORT_PRIO_TYPE uXPriority_) {
00     return &m_aclPriorities[uXPriority_]; }
00150     static ThreadList* GetStopList() { return &m_clStopList; }
00159     static bool IsEnabled() { return m_bEnabled; }
00166     static void QueueScheduler() { m_bQueuedSchedule = true; }
00167 private:
00169     static bool m_bEnabled;
00170
00172     static bool m_bQueuedSchedule;
00173
00175     static ThreadList m_clStopList;
00176
00178     static ThreadList m_aclPriorities[
00     KERNEL_NUM_PRIORITIES];
00179
00181     static PriorityMap m_clPrioMap;
00182 };
00183 #endif
```

20.97 /home/moslevin/mark3-source/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"
#include "autoalloc.h"
#include "priomap.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.

20.97.1 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.

20.98 thread.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__    |    _|__    |    _|__    |    _|__    |__    _____
00004 |    \  /    |  |       \    |    |    |       ||  /  /     ||___   |
00005 |     \/     |  |        \    |    |    |       ||     |     ||___   |
00006 |__/\__/|__|_||__|\__\   _||__|\__\   _||__|\__\   _||_____|
00007     |____|       |____|       |____|       |____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 #include "autoalloc.h"
00047 #include "priomap.h"
00048
00049 class Thread;
00050
```

```
00051 //---------------------------------------------------------------------------
00052 typedef void (*ThreadCreateCallout_t)(Thread* pclThread_);
00053 typedef void (*ThreadExitCallout_t)(Thread* pclThread_);
00054 typedef void (*ThreadContextCallout_t)(Thread* pclThread_);
00055
00056 //---------------------------------------------------------------------------
00060 class Thread : public LinkListNode
00061 {
00062 public:
00063     void* operator new(size_t sz, void* pv) { return (Thread*)pv; };
00064     ~Thread();
00065
00081     void
00082     Init(K_WORD* pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_,
     ThreadEntry_t pfEntryPoint_, void* pvArg_);
00083
00084 #if KERNEL_USE_AUTO_ALLOC
00085
00103     static Thread* Init(uint16_t u16StackSize_, uint8_t uXPriority_,
     ThreadEntry_t pfEntryPoint_, void* pvArg_);
00104 #endif
00105
00113     void Start();
00114
00121     void Stop();
00122
00123 #if KERNEL_USE_THREADNAME
00124
00133     void SetName(const char* szName_) { m_szName = szName_; }
00140     const char* GetName() { return m_szName; }
00141 #endif
00142
00151     ThreadList* GetOwner(void) { return m_pclOwner; }
00159     ThreadList* GetCurrent(void) { return m_pclCurrent; }
00168     PORT_PRIO_TYPE GetPriority(void) { return
     m_uXPriority; }
00176     PORT_PRIO_TYPE GetCurPriority(void) { return
     m_uXCurPriority; }
00177 #if KERNEL_USE_QUANTUM
00178
00185     void SetQuantum(uint16_t u16Quantum_) { m_u16Quantum = u16Quantum_; }
00193     uint16_t GetQuantum(void) { return m_u16Quantum; }
00194 #endif
00195
00203     void SetCurrent(ThreadList* pclNewList_) { m_pclCurrent = pclNewList_;
     }
00211     void SetOwner(ThreadList* pclNewList_) { m_pclOwner = pclNewList_; }
00224     void SetPriority(PORT_PRIO_TYPE uXPriority_);
00225
00235     void InheritPriority(PORT_PRIO_TYPE uXPriority_);
00236
00237 #if KERNEL_USE_DYNAMIC_THREADS
00238
00249     void Exit();
00250 #endif
00251
00252 #if KERNEL_USE_SLEEP
00253
00261     static void Sleep(uint32_t u32TimeMs_);
00262
00271     static void USleep(uint32_t u32TimeUs_);
00272 #endif
00273
00281     static void Yield(void);
00282
00290     void SetID(uint8_t u8ID_) { m_u8ThreadID = u8ID_; }
00298     uint8_t GetID() { return m_u8ThreadID; }
00311     uint16_t GetStackSlack();
00312
00313 #if KERNEL_USE_EVENTFLAG
00314
00321     uint16_t GetEventFlagMask() { return m_u16FlagMask; }
00326     void SetEventFlagMask(uint16_t u16Mask_) { m_u16FlagMask = u16Mask_; }
00332     void SetEventFlagMode(EventFlagOperation_t eMode_) {
     m_eFlagMode = eMode_; }
00337     EventFlagOperation_t GetEventFlagMode() { return
     m_eFlagMode; }
00338 #endif
00339
00340 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00341
00344     Timer* GetTimer();
00345 #endif
00346 #if KERNEL_USE_TIMEOUTS
00347
00355     void SetExpired(bool bExpired_);
00356
```

```
00363      bool GetExpired();
00364 #endif
00365
00366 #if KERNEL_USE_IDLE_FUNC
00367
00372      void InitIdle();
00373 #endif
00374
00381      ThreadState_t GetState() { return m_eState; }
00389      void SetState(ThreadState_t eState_) { m_eState = eState_; }
00390      friend class ThreadPort;
00391
00392 private:
00400      static void ContextSwitchSWI(void);
00401
00407      void SetPriorityBase(PORT_PRIO_TYPE uXPriority_);
00408
00410      K_WORD* m_pwStackTop;
00411
00413      K_WORD* m_pwStack;
00414
00416      uint8_t m_u8ThreadID;
00417
00419      PORT_PRIO_TYPE m_uXPriority;
00420
00422      PORT_PRIO_TYPE m_uXCurPriority;
00423
00425      ThreadState_t m_eState;
00426
00427 #if KERNEL_USE_THREADNAME
00428      const char* m_szName;
00430 #endif
00431
00433      uint16_t m_u16StackSize;
00434
00436      ThreadList* m_pclCurrent;
00437
00439      ThreadList* m_pclOwner;
00440
00442      ThreadEntry_t m_pfEntryPoint;
00443
00445      void* m_pvArg;
00446
00447 #if KERNEL_USE_QUANTUM
00448      uint16_t m_u16Quantum;
00450 #endif
00451
00452 #if KERNEL_USE_EVENTFLAG
00453      uint16_t m_u16FlagMask;
00455
00457      EventFlagOperation_t m_eFlagMode;
00458 #endif
00459
00460 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00461      Timer m_clTimer;
00463 #endif
00464 #if KERNEL_USE_TIMEOUTS
00465      bool m_bExpired;
00467 #endif
00468 };
00469
00470 #if KERNEL_USE_IDLE_FUNC
00471 //---------------------------------------------------------------------------
00483 typedef struct {
00484      LinkListNode* next;
00485      LinkListNode* prev;
00486
00488      K_WORD* m_pwStackTop;
00489
00491      K_WORD* m_pwStack;
00492
00494      uint8_t m_u8ThreadID;
00495
00497      PORT_PRIO_TYPE m_uXPriority;
00498
00500      PORT_PRIO_TYPE m_uXCurPriority;
00501
00503      ThreadState_t m_eState;
00504
00505 #if KERNEL_USE_THREADNAME
00506      const char* m_szName;
00508 #endif
00509
00510 } FakeThread_t;
00511 #endif
00512
00513 #endif
```

20.99 /home/moslevin/mark3-source/embedded/kernel/public/threadlist.h File Reference

Thread linked-list declarations.

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

Classes

- class ThreadList

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

20.99.1 Detailed Description

Thread linked-list declarations.

Definition in file threadlist.h.

20.100 threadlist.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /   |  | ||    \       ||        ||  |/ /      ||___   |
00005 |     \/    |  | ||     \      ||    \    ||   \       ||__    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||____|
00007      |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 "priomap.h"
00027 #include "ll.h"
00028
00029 class Thread;
00030
00035 class ThreadList : public CircularLinkList
00036 {
00037 public:
00038     void* operator new(size_t sz, void* pv) { return (ThreadList*)pv; };
00044     ThreadList() : m_uXPriority(0), m_pclMap(NULL)
00045     { }
00046
00054     void SetPriority(PORT_PRIO_TYPE uXPriority_);
00055
00065     void SetMapPointer(PriorityMap* pclMap_);
00066
00074     void Add(LinkListNode* node_);
00075
00087     void Add(LinkListNode* node_, PriorityMap* pclMap_,
    PORT_PRIO_TYPE uXPriority_);
00088
00097     void AddPriority(LinkListNode* node_);
00098
00106     void Remove(LinkListNode* node_);
00107
00115     Thread* HighestWaiter();
00116
00117 private:
00119     PORT_PRIO_TYPE m_uXPriority;
00120
00122     PriorityMap* m_pclMap;
00123 };
00124
00125 #endif
```

20.101 /home/moslevin/mark3-source/embedded/kernel/public/timer.h File Reference

Timer object declarations.

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

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.

20.101.1 Detailed Description

Timer object declarations.

Definition in file timer.h.

20.101.2 Macro Definition Documentation

20.101.2.1 #define TIMERLIST_FLAG_EXPIRED (0x08)

Timer is actually expired.

Definition at line 36 of file timer.h.

20.101.3 Typedef Documentation

20.101.3.1 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.

20.102 timer.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__   _|__  |__    _____
00004 |    \  /  |  | |     \      ||      |      ||  |/ /      ||___     |
00005 |     \/   |  | |      \     ||      |      ||  |\   \    ||___     |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007      |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 #define TIMER_TICKS_INVALID (0x80000000)
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) * PORT_TIMER_FREQ))
00059 #define MSECONDS_TO_TICKS(x) ((((((uint32_t)x) * (PORT_TIMER_FREQ / 100)) + 5) / 10))
00060 #define USECONDS_TO_TICKS(x) ((((((uint32_t)x) * PORT_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;
00097
00098 class Timer : public LinkListNode
00099 {
00100 public:
00101     void* operator new(size_t sz, void* pv) { return (Timer*)pv; };
```

```
00108     Timer() { m_u8Flags = 0; }
00114     void Init()
00115     {
00116         ClearNode();
00117         m_u32Interval      = 0;
00118         m_u32TimerTolerance = 0;
00119         m_u32TimeLeft       = 0;
00120         m_u8Flags          = 0;
00121     }
00122
00134     void Start(bool bRepeat_, uint32_t u32IntervalMs_, TimerCallback_t pfCallback_, void*
      pvData_);
00135
00149     void
00150     Start(bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_,
      TimerCallback_t pfCallback_, void* pvData_);
00151
00160     void Start();
00161
00168     void Stop();
00169
00179     void SetFlags(uint8_t u8Flags_) { m_u8Flags = u8Flags_; }
00187     void SetCallback(TimerCallback_t pfCallback_) { m_pfCallback = pfCallback_; }
00195     void SetData(void* pvData_) { m_pvData = pvData_; }
00204     void SetOwner(Thread* pclOwner_) { m_pclOwner = pclOwner_; }
00212     void SetIntervalTicks(uint32_t u32Ticks_);
00213
00221     void SetIntervalSeconds(uint32_t u32Seconds_);
00222
00230     uint32_t GetInterval() { return m_u32Interval; }
00238     void SetIntervalMSeconds(uint32_t u32MSeconds_);
00239
00247     void SetIntervalUSeconds(uint32_t u32USeconds_);
00248
00257     void SetTolerance(uint32_t u32Ticks_);
00258
00259 private:
00260     friend class TimerList;
00261
00263     uint8_t m_u8Flags;
00264
00266     TimerCallback_t m_pfCallback;
00267
00269     uint32_t m_u32Interval;
00270
00272     uint32_t m_u32TimeLeft;
00273
00275     uint32_t m_u32TimerTolerance;
00276
00278     Thread* m_pclOwner;
00279
00281     void* m_pvData;
00282 };
00283
00284 #endif // KERNEL_USE_TIMERS
00285
00286 #endif
```

20.103   /home/moslevin/mark3-source/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.

20.103.1 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.

20.104 timerlist.h

```
00001 /*=============================================================================
00002        _____        _____        _____        _____
00003  ___| _|_ __| __|_      |__ __|__    |__ __|__    |__   _____
00004 |      \  /   | ||      \      ||      |      || |/ /      ||___   |
00005 |       \/    | ||      \      ||      \      ||      \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\   __||__|\__\  __||_____|
00007       |____|       |____|        |____|        |____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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
```

20.105 /home/moslevin/mark3-source/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.

20.105.1 Detailed Description

Timer scheduler declarations.

Definition in file timerscheduler.h.

20.106 timerscheduler.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /    | ||    \     ||     |     ||   |/ /     ||___  |
00005 |     \/     | ||     \    ||     \     ||   |  \     ||__   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007    |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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(); }
00056     static void Add(Timer* pclListNode_) { m_clTimerList.Add(pclListNode_); }
00065     static void Remove(Timer* pclListNode_) { m_clTimerList.
00071     Remove(pclListNode_); }
00074     static void Process() { m_clTimerList.Process(); }
00075 private:
00077     static TimerList m_clTimerList;
00078 };
00079
00080 #endif // KERNEL_USE_TIMERS
00081
00082 #endif //__TIMERSCHEDULER_H__
```

20.107 /home/moslevin/mark3-source/embedded/kernel/public/tracebuffer.h File Reference

Kernel trace buffer class declaration.

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

20.107.1 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.

20.108 tracebuffer.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /   | ||    \      ||      |     ||    |/ /    ||___  |
00005 |     \/    | ||     \     ||      \     ||    |  \    ||___   |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007      |____|        |____|        |____|        |____|
00008
00009 --[Mark3 Realtime Platform]------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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
00030 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00031
00032 #define TRACE_BUFFER_SIZE (160)
00033
00034 typedef void (*TraceBufferCallback_t)(uint16_t* pu16Source_, uint16_t u16Len_, bool bPingPong_);
00035
00039 class TraceBuffer
00040 {
00041 public:
00047     static void Init();
00048
00053     static uint16_t Increment(void) { return m_u16SyncNumber++; }
00062     static void Write(uint16_t* pu16Data_, uint16_t u16Size_);
00063
00072     static void SetCallback(TraceBufferCallback_t pfCallback_) { m_pfCallback = pfCallback_; }
00073 private:
00074     static TraceBufferCallback_t m_pfCallback;
00075     static uint16_t             m_u16SyncNumber;
00076     static uint16_t             m_u16Index;
00077     static uint16_t             m_au16Buffer[(TRACE_BUFFER_SIZE / sizeof(uint16_t))];
00078 };
00079
00080 #endif // KERNEL_USE_DEBUG
00081
00082 #endif
```

20.109 /home/moslevin/mark3-source/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.

20.109.1 Detailed Description

Thread Quantum Implementation for Round-Robin Scheduling.

Definition in file quantum.cpp.

20.109.2 Function Documentation

20.109.2.1 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.

20.110 quantum.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__    _|__  __|_    |__    |__    _____
00004 |    \  /  |  |  | |    \      |  | |    |      |  | |/ /       | |___  |
00005 |     \/   |  |  | |     \     |  | |    |      |  | |    \      | |___   |
00006 |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007     |____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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          if (pclThread_->GetCurrent()->GetHead() != pclThread_->
      GetCurrent()->GetTail()) {
00069              bAddQuantumTimer = true;
00070              pclThread_->GetCurrent()->PivotForward();
00071          }
00072      }
00073 }
00074
00075 //---------------------------------------------------------------------------
00076 void Quantum::SetTimer(Thread* pclThread_)
00077 {
00078      m_clQuantumTimer.SetIntervalMSeconds(pclThread_->GetQuantum());
00079      m_clQuantumTimer.SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00080      m_clQuantumTimer.SetData(NULL);
00081      m_clQuantumTimer.SetCallback((TimerCallback_t)QuantumCallback);
00082      m_clQuantumTimer.SetOwner(pclThread_);
00083 }
00084
00085 //---------------------------------------------------------------------------
00086 void Quantum::AddThread(Thread* pclThread_)
00087 {
00088      if (m_bActive
00089 #if KERNEL_USE_IDLE_FUNC
00090          || (pclThread_ == Kernel::GetIdleThread())
00091 #endif
00092              ) {
00093          return;
00094      }
00095
00096      // If this is called from the timer callback, queue a timer add...
00097      if (m_bInTimer) {
00098          bAddQuantumTimer = true;
00099          return;
00100      }
00101
00102      // If this isn't the only thread in the list.
00103      if (pclThread_->GetCurrent()->GetHead() != pclThread_->
      GetCurrent()->GetTail()) {
00104          Quantum::SetTimer(pclThread_);
00105          TimerScheduler::Add(&m_clQuantumTimer);
00106          m_bActive = 1;
00107      }
00108 }
00109
00110 //---------------------------------------------------------------------------
00111 void Quantum::RemoveThread(void)
00112 {
00113      if (!m_bActive) {
00114          return;
00115      }
00116
00117      // Cancel the current timer
00118      TimerScheduler::Remove(&m_clQuantumTimer);
00119      m_bActive = 0;
00120 }
00121
00122 //---------------------------------------------------------------------------
00123 void Quantum::UpdateTimer(void)
00124 {
00125      // If we have to re-add the quantum timer (more than 2 threads at the
00126      // high-priority level...)
00127      if (bAddQuantumTimer) {
00128          // Trigger a thread yield - this will also re-schedule the
00129          // thread *and* reset the round-robin scheduler.
00130          Thread::Yield();
00131          bAddQuantumTimer = false;
00132      }
00133 }
00134
00135 #endif // KERNEL_USE_QUANTUM
```

20.111 /home/moslevin/mark3-source/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.

20.111.1 Detailed Description

Strict-Priority + Round-Robin thread scheduler implementation.

Definition in file scheduler.cpp.

20.112 scheduler.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  |__    _|__  |__    _|  __  |__    _____
00004 |    \  /    | | |     \       | |        | |  |/ /        | |___       |
00005 |     \/     | | |      \      | |    __   | |  |  \        | |__        |
00006 |__/\__/|__|_| |__|\__\   _|__|__|\__\   _|__|__|\__\   __|__|____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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 //---------------------------------------------------------------------------
00049 ThreadList  Scheduler::m_clStopList;
00050 ThreadList  Scheduler::m_aclPriorities[
     KERNEL_NUM_PRIORITIES];
```

```
00051 PriorityMap Scheduler::m_clPrioMap;
00052
00053 //---------------------------------------------------------------------------
00054 void Scheduler::Init()
00055 {
00056     for (int i = 0; i < KERNEL_NUM_PRIORITIES; i++) {
00057         m_aclPriorities[i].SetPriority(i);
00058         m_aclPriorities[i].SetMapPointer(&
     m_clPrioMap);
00059     }
00060 }
00061
00062 //---------------------------------------------------------------------------
00063 void Scheduler::Schedule()
00064 {
00065     PORT_PRIO_TYPE uXPrio;
00066
00067     uXPrio = m_clPrioMap.HighestPriority();
00068
00069 #if KERNEL_USE_IDLE_FUNC
00070     if (uXPrio == 0) {
00071         // There aren't any active threads at all - set g_pclNext to IDLE
00072         g_pclNext = Kernel::GetIdleThread();
00073     } else
00074 #endif
00075     {
00076         if (uXPrio == 0) {
00077             Kernel::Panic(PANIC_NO_READY_THREADS);
00078         }
00079         // Priorities are one-indexed
00080         uXPrio--;
00081
00082         // Get the thread node at this priority.
00083         g_pclNext = (Thread*)(m_aclPriorities[uXPrio].GetHead());
00084     }
00085     KERNEL_TRACE_1("Next Thread: %d\n", (uint16_t)((Thread*)g_pclNext)->GetID());
00086 }
00087
00088 //---------------------------------------------------------------------------
00089 void Scheduler::Add(Thread* pclThread_)
00090 {
00091     m_aclPriorities[pclThread_->GetPriority()].Add(pclThread_);
00092 }
00093
00094 //---------------------------------------------------------------------------
00095 void Scheduler::Remove(Thread* pclThread_)
00096 {
00097     m_aclPriorities[pclThread_->GetPriority()].Remove(pclThread_);
00098 }
00099
00100 //---------------------------------------------------------------------------
00101 bool Scheduler::SetScheduler(bool bEnable_)
00102 {
00103     bool bRet;
00104     CS_ENTER();
00105     bRet      = m_bEnabled;
00106     m_bEnabled = bEnable_;
00107     // If there was a queued scheduler evevent, dequeue and trigger an
00108     // immediate Yield
00109     if (m_bEnabled && m_bQueuedSchedule) {
00110         m_bQueuedSchedule = false;
00111         Thread::Yield();
00112     }
00113     CS_EXIT();
00114     return bRet;
00115 }
```

20.113 /home/moslevin/mark3-source/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 "priomap.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.

20.113.1 Detailed Description

Platform-Independent thread class Definition.

Definition in file thread.cpp.

20.114 thread.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003 ___|    _|__    __|_    |__    __|_    |__    __|_    |__    |__    _____
00004 |     \ /  |   | ||       \    ||      ||   |/ /      ||___   |
00005 |      \/   |   | ||        \      ||      \     ||    \       ||__    |
00006 |__/\__/|__|__||__|\__\   _||__|\__\   _||__|\__\   _||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #include "kerneltypes.h"
00023 #include "mark3cfg.h"
00024
00025 #include "thread.h"
00026 #include "scheduler.h"
00027 #include "kernelswi.h"
00028 #include "timerlist.h"
00029 #include "ksemaphore.h"
00030 #include "quantum.h"
00031 #include "kernel.h"
00032 #include "priomap.h"
00033
00034 #define _CAN_HAS_DEBUG
00035 //--[Autogenerated - Do Not Modify]-----------------------------------------
00036 #include "dbg_file_list.h"
00037 #include "buffalogger.h"
00038 #if defined(DBG_FILE)
00039 #error "Debug logging file token already defined!  Bailing."
00040 #else
00041 #define DBG_FILE _DBG___KERNEL_THREAD_CPP
00042 #endif
00043 //--[End Autogenerated content]---------------------------------------------
00044
00045 #include "kerneldebug.h"
00046 //---------------------------------------------------------------------------
00047 Thread::~Thread()
00048 {
00049     // On destruction of a thread located on a stack,
```

```
00050      // ensure that the thread is either stopped, or exited.
00051      // If the thread is stopped, move it to the exit state.
00052      // If not in the exit state, kernel panic -- it's catastrophic to have
00053      // running threads on stack suddenly disappear.
00054      if (m_eState == THREAD_STATE_STOP) {
00055          CS_ENTER();
00056          m_pclCurrent->Remove(this);
00057          m_pclCurrent = 0;
00058          m_pclOwner   = 0;
00059          m_eState     = THREAD_STATE_EXIT;
00060          CS_EXIT();
00061      } else if (m_eState != THREAD_STATE_EXIT) {
00062 #if KERNEL_AWARE_SIMULATION
00063          KernelAware::Trace(0, 0, m_u8ThreadID,
00     m_eState);
00064 #endif
00065          Kernel::Panic(PANIC_RUNNING_THREAD_DESCOPED);
00066      }
00067 }
00068
00069 //---------------------------------------------------------------------------
00070 void Thread::Init(
00071      K_WORD* pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_,
00     ThreadEntry_t pfEntryPoint_, void* pvArg_)
00072 {
00073      static uint8_t u8ThreadID = 0;
00074
00075      KERNEL_ASSERT(pwStack_);
00076      KERNEL_ASSERT(pfEntryPoint_);
00077
00078      ClearNode();
00079
00080      m_u8ThreadID = u8ThreadID++;
00081 #if KERNEL_USE_IDLE_FUNC
00082      if (u8ThreadID == 255) {
00083          u8ThreadID = 0;
00084      }
00085 #endif
00086
00087      KERNEL_TRACE_1("Stack Size: %d", u16StackSize_);
00088      KERNEL_TRACE_1("Thread Pri: %d", (uint8_t)uXPriority_);
00089      KERNEL_TRACE_1("Thread Id: %d", (uint16_t)m_u8ThreadID);
00090      KERNEL_TRACE_1("Entrypoint: %x", (uint16_t)pfEntryPoint_);
00091
00092      // Initialize the thread parameters to their initial values.
00093      m_pwStack    = pwStack_;
00094      m_pwStackTop = TOP_OF_STACK(pwStack_, u16StackSize_);
00095
00096      m_u16StackSize = u16StackSize_;
00097
00098 #if KERNEL_USE_QUANTUM
00099      m_u16Quantum = THREAD_QUANTUM_DEFAULT;
00100 #endif
00101
00102      m_uXPriority    = uXPriority_;
00103      m_uXCurPriority = m_uXPriority;
00104      m_pfEntryPoint  = pfEntryPoint_;
00105      m_pvArg         = pvArg_;
00106      m_eState        = THREAD_STATE_STOP;
00107
00108 #if KERNEL_USE_THREADNAME
00109      m_szName = NULL;
00110 #endif
00111 #if KERNEL_USE_TIMERS
00112      m_clTimer.Init();
00113 #endif
00114
00115      // Call CPU-specific stack initialization
00116      ThreadPort::InitStack(this);
00117
00118      // Add to the global "stop" list.
00119      CS_ENTER();
00120      m_pclOwner   = Scheduler::GetThreadList(
00     m_uXPriority);
00121      m_pclCurrent = Scheduler::GetStopList();
00122      m_pclCurrent->Add(this);
00123      CS_EXIT();
00124
00125 #if KERNEL_USE_THREAD_CALLOUTS
00126      ThreadCreateCallout_t pfCallout = Kernel::GetThreadCreateCallout();
00127      if (pfCallout) {
00128          pfCallout(this);
00129      }
00130 #endif
00131 }
00132
00133 #if KERNEL_USE_AUTO_ALLOC
```

```
00134 //---------------------------------------------------------------------------
00135 Thread* Thread::Init(uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_,
       ThreadEntry_t pfEntryPoint_, void* pvArg_)
00136 {
00137     Thread* pclNew  = (Thread*)AutoAlloc::Allocate(sizeof(Thread));
00138     K_WORD* pwStack = (K_WORD*)AutoAlloc::Allocate(u16StackSize_);
00139     pclNew->Init(pwStack, u16StackSize_, uXPriority_, pfEntryPoint_, pvArg_);
00140     return pclNew;
00141 }
00142 #endif
00143
00144 //---------------------------------------------------------------------------
00145 void Thread::Start(void)
00146 {
00147     // Remove the thread from the scheduler's "stopped" list, and add it
00148     // to the scheduler's ready list at the proper priority.
00149     KERNEL_TRACE_1("Starting Thread %d", (uint16_t)m_u8ThreadID);
00150
00151     CS_ENTER();
00152     Scheduler::GetStopList()->Remove(this);
00153     Scheduler::Add(this);
00154     m_pclOwner   = Scheduler::GetThreadList(
       m_uXPriority);
00155     m_pclCurrent = m_pclOwner;
00156     m_eState     = THREAD_STATE_READY;
00157
00158 #if KERNEL_USE_QUANTUM
00159     if (Kernel::IsStarted()) {
00160         if (GetCurPriority() >= Scheduler::GetCurrentThread()->
       GetCurPriority()) {
00161             // Deal with the thread Quantum
00162             Quantum::RemoveThread();
00163             Quantum::AddThread(this);
00164         }
00165     }
00166 #endif
00167
00168     if (Kernel::IsStarted()) {
00169         if (GetCurPriority() >= Scheduler::GetCurrentThread()->
       GetCurPriority()) {
00170             Thread::Yield();
00171         }
00172     }
00173     CS_EXIT();
00174 }
00175
00176 //---------------------------------------------------------------------------
00177 void Thread::Stop()
00178 {
00179     bool bReschedule = 0;
00180
00181     CS_ENTER();
00182
00183     // If a thread is attempting to stop itself, ensure we call the scheduler
00184     if (this == Scheduler::GetCurrentThread()) {
00185         bReschedule = true;
00186     }
00187
00188     // Add this thread to the stop-list (removing it from active scheduling)
00189     // Remove the thread from scheduling
00190     if (m_eState == THREAD_STATE_READY) {
00191         Scheduler::Remove(this);
00192     } else if (m_eState == THREAD_STATE_BLOCKED) {
00193         m_pclCurrent->Remove(this);
00194     }
00195
00196     m_pclOwner   = Scheduler::GetStopList();
00197     m_pclCurrent = m_pclOwner;
00198     m_pclOwner->Add(this);
00199     m_eState = THREAD_STATE_STOP;
00200
00201 #if KERNEL_USE_TIMERS
00202     // Just to be safe - attempt to remove the thread's timer
00203     // from the timer-scheduler (does no harm if it isn't
00204     // in the timer-list)
00205     TimerScheduler::Remove(&m_clTimer);
00206 #endif
00207
00208     CS_EXIT();
00209
00210     if (bReschedule) {
00211         Thread::Yield();
00212     }
00213 }
00214
00215 #if KERNEL_USE_DYNAMIC_THREADS
00216 //---------------------------------------------------------------------------
```

```
00217 void Thread::Exit()
00218 {
00219     bool bReschedule = 0;
00220
00221     KERNEL_TRACE_1("Exit Thread %d", m_u8ThreadID);
00222
00223 #if KERNEL_USE_THREAD_CALLOUTS
00224     ThreadExitCallout_t pfCallout = Kernel::GetThreadExitCallout();
00225     if (pfCallout) {
00226         pfCallout(this);
00227     }
00228 #endif
00229
00230     CS_ENTER();
00231
00232     // If this thread is the actively-running thread, make sure we run the
00233     // scheduler again.
00234     if (this == Scheduler::GetCurrentThread()) {
00235         bReschedule = 1;
00236     }
00237
00238     // Remove the thread from scheduling
00239     if (m_eState == THREAD_STATE_READY) {
00240         Scheduler::Remove(this);
00241     } else if ((m_eState == THREAD_STATE_BLOCKED) || (m_eState == THREAD_STATE_STOP)) {
00242         m_pclCurrent->Remove(this);
00243     }
00244
00245     m_pclCurrent = 0;
00246     m_pclOwner   = 0;
00247     m_eState     = THREAD_STATE_EXIT;
00248
00249     // We've removed the thread from scheduling, but interrupts might
00250     // trigger checks against this thread's currently priority before
00251     // we get around to scheduling new threads.  As a result, set the
00252     // priority to idle to ensure that we always wind up scheduling
00253     // new threads.
00254     m_uXCurPriority = 0;
00255     m_uXPriority    = 0;
00256
00257 #if KERNEL_USE_TIMERS
00258     // Just to be safe - attempt to remove the thread's timer
00259     // from the timer-scheduler (does no harm if it isn't
00260     // in the timer-list)
00261     TimerScheduler::Remove(&m_clTimer);
00262 #endif
00263
00264     CS_EXIT();
00265
00266     if (bReschedule) {
00267         // Choose a new "next" thread if we must
00268         Thread::Yield();
00269     }
00270 }
00271 #endif
00272
00273 #if KERNEL_USE_SLEEP
00274 //---------------------------------------------------------------------------
00276 static void ThreadSleepCallback(Thread* pclOwner_, void* pvData_)
00277 {
00278     Semaphore* pclSemaphore = static_cast<Semaphore*>(pvData_);
00279     // Post the semaphore, which will wake the sleeping thread.
00280     pclSemaphore->Post();
00281 }
00282
00283 //---------------------------------------------------------------------------
00284 void Thread::Sleep(uint32_t u32TimeMs_)
00285 {
00286     Semaphore clSemaphore;
00287     Timer*    pclTimer = g_pclCurrent->GetTimer();
00288
00289     // Create a semaphore that this thread will block on
00290     clSemaphore.Init(0, 1);
00291
00292     // Create a one-shot timer that will call a callback that posts the
00293     // semaphore, waking our thread.
00294     pclTimer->Init();
00295     pclTimer->SetIntervalMSeconds(u32TimeMs_);
00296     pclTimer->SetCallback(ThreadSleepCallback);
00297     pclTimer->SetData((void*)&clSemaphore);
00298     pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00299
00300     // Add the new timer to the timer scheduler, and block the thread
00301     TimerScheduler::Add(pclTimer);
00302     clSemaphore.Pend();
00303 }
00304
```

```
00305 //---------------------------------------------------------------------------
00306 void Thread::USleep(uint32_t u32TimeUs_)
00307 {
00308     Semaphore clSemaphore;
00309     Timer*    pclTimer = g_pclCurrent->GetTimer();
00310
00311     // Create a semaphore that this thread will block on
00312     clSemaphore.Init(0, 1);
00313
00314     // Create a one-shot timer that will call a callback that posts the
00315     // semaphore, waking our thread.
00316     pclTimer->Init();
00317     pclTimer->SetIntervalUSeconds(u32TimeUs_);
00318     pclTimer->SetCallback(ThreadSleepCallback);
00319     pclTimer->SetData((void*)&clSemaphore);
00320     pclTimer->SetFlags(TIMERLIST_FLAG_ONE_SHOT);
00321
00322     // Add the new timer to the timer scheduler, and block the thread
00323     TimerScheduler::Add(pclTimer);
00324     clSemaphore.Pend();
00325 }
00326 #endif // KERNEL_USE_SLEEP
00327
00328 //---------------------------------------------------------------------------
00329 uint16_t Thread::GetStackSlack()
00330 {
00331     K_ADDR wTop    = (K_ADDR)m_u16StackSize - 1;
00332     K_ADDR wBottom = (K_ADDR)0;
00333     K_ADDR wMid    = ((wTop + wBottom) + 1) / 2;
00334
00335     CS_ENTER();
00336
00337     // Logarithmic bisection - find the point where the contents of the
00338     // stack go from 0xFF's to non 0xFF.  Not Definitive, but accurate enough
00339     while ((wTop - wBottom) > 1) {
00340 #if STACK_GROWS_DOWN
00341         if (m_pwStack[wMid] != (K_WORD)(-1))
00342 #else
00343         if (m_pwStack[wMid] == (K_WORD)(-1))
00344 #endif
00345         {
00346             wTop = wMid;
00347         } else {
00348             wBottom = wMid;
00349         }
00350         wMid = (wTop + wBottom + 1) / 2;
00351     }
00352
00353     CS_EXIT();
00354
00355     return wMid;
00356 }
00357 }
00358
00359 //---------------------------------------------------------------------------
00360 void Thread::Yield()
00361 {
00362     CS_ENTER();
00363     // Run the scheduler
00364     if (Scheduler::IsEnabled()) {
00365         Scheduler::Schedule();
00366
00367         // Only switch contexts if the new task is different than the old task
00368         if (Scheduler::GetCurrentThread() !=
00369     Scheduler::GetNextThread()) {
00369 #if KERNEL_USE_QUANTUM
00370             // new thread scheduled.  Stop current quantum timer (if it exists),
00371             // and restart it for the new thread (if required).
00372             Quantum::RemoveThread();
00373             Quantum::AddThread((Thread*)g_pclNext);
00374 #endif
00375             Thread::ContextSwitchSWI();
00376         }
00377     } else {
00378         Scheduler::QueueScheduler();
00379     }
00380
00381     CS_EXIT();
00382 }
00383
00384 //---------------------------------------------------------------------------
00385 void Thread::SetPriorityBase(PORT_PRIO_TYPE uXPriority_)
00386 {
00387     GetCurrent()->Remove(this);
00388
00389     SetCurrent(Scheduler::GetThreadList(
00389     m_uXPriority));
00390
```

```
00391     GetCurrent()->Add(this);
00392 }
00393
00394 //---------------------------------------------------------------------------
00395 void Thread::SetPriority(PORT_PRIO_TYPE uXPriority_)
00396 {
00397     bool bSchedule = 0;
00398
00399     CS_ENTER();
00400     // If this is the currently running thread, it's a good idea to reschedule
00401     // Or, if the new priority is a higher priority than the current thread's.
00402     if ((g_pclCurrent == this) || (uXPriority_ > g_pclCurrent->
    GetPriority())) {
00403         bSchedule = 1;
00404     }
00405     Scheduler::Remove(this);
00406     CS_EXIT();
00407
00408     m_uXCurPriority = uXPriority_;
00409     m_uXPriority    = uXPriority_;
00410
00411     CS_ENTER();
00412     Scheduler::Add(this);
00413     CS_EXIT();
00414
00415     if (bSchedule) {
00416         if (Scheduler::IsEnabled()) {
00417             CS_ENTER();
00418             Scheduler::Schedule();
00419 #if KERNEL_USE_QUANTUM
00420             // new thread scheduled.  Stop current quantum timer (if it exists),
00421             // and restart it for the new thread (if required).
00422             Quantum::RemoveThread();
00423             Quantum::AddThread((Thread*)g_pclNext);
00424 #endif
00425             CS_EXIT();
00426             Thread::ContextSwitchSWI();
00427         } else {
00428             Scheduler::QueueScheduler();
00429         }
00430     }
00431 }
00432
00433 //---------------------------------------------------------------------------
00434 void Thread::InheritPriority(PORT_PRIO_TYPE uXPriority_)
00435 {
00436     SetOwner(Scheduler::GetThreadList(uXPriority_));
00437     m_uXCurPriority = uXPriority_;
00438 }
00439
00440 //---------------------------------------------------------------------------
00441 void Thread::ContextSwitchSWI()
00442 {
00443     // Call the context switch interrupt if the scheduler is enabled.
00444     if (Scheduler::IsEnabled() == 1) {
00445         KERNEL_TRACE_1("Context switch to Thread %d", (uint16_t)((
    Thread*)g_pclNext)->GetID());
00446 #if KERNEL_USE_STACK_GUARD
00447         uint16_t u16Slack;
00448 #if KERNEL_USE_IDLE_FUNC
00449         if (g_pclCurrent->GetID() != 255) {
00450 #endif
00451             if (g_pclCurrent->GetStackSlack() <= Kernel::GetStackGuardThreshold())
     {
00452                 KernelAware::Trace(DBG_FILE, __LINE__,
    g_pclCurrent->GetID(), g_pclCurrent->GetStackSlack());
00453                 Kernel::Panic(PANIC_STACK_SLACK_VIOLATED);
00454             }
00455 #if KERNEL_USE_IDLE_FUNC
00456         }
00457 #endif
00458 #endif
00459
00460 #if KERNEL_USE_THREAD_CALLOUTS
00461         ThreadContextCallout_t pfCallout = Kernel::GetThreadContextSwitchCallout
    ();
00462         if (pfCallout) {
00463             pfCallout(g_pclCurrent);
00464         }
00465 #endif
00466         KernelSWI::Trigger();
00467     }
00468 }
00469
00470 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00471 //---------------------------------------------------------------------------
00472 Timer* Thread::GetTimer()
```

```
00473 {
00474     return &m_clTimer;
00475 }
00476 #endif
00477 #if KERNEL_USE_TIMEOUTS
00478 //---------------------------------------------------------------------------
00479 void Thread::SetExpired(bool bExpired_)
00480 {
00481     m_bExpired = bExpired_;
00482 }
00483
00484 //---------------------------------------------------------------------------
00485 bool Thread::GetExpired()
00486 {
00487     return m_bExpired;
00488 }
00489 #endif
00490
00491 #if KERNEL_USE_IDLE_FUNC
00492 //---------------------------------------------------------------------------
00493 void Thread::InitIdle(void)
00494 {
00495     ClearNode();
00496
00497     m_uXPriority    = 0;
00498     m_uXCurPriority = 0;
00499     m_pfEntryPoint  = 0;
00500     m_pvArg         = 0;
00501     m_u8ThreadID    = 255;
00502     m_eState        = THREAD_STATE_READY;
00503 #if KERNEL_USE_THREADNAME
00504     m_szName = "IDLE";
00505 #endif
00506 }
00507 #endif
```

20.115 /home/moslevin/mark3-source/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"
```

20.115.1 Detailed Description

Thread linked-list definitions.

Definition in file threadlist.cpp.

20.116 threadlist.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__   _|__  __|_  _____
00004 |    \  /    | ||    \        | |     | |   |/ /    | |___   |
00005 |     \/     | ||     \       | |      \     | |    \      | |__    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||____|
00007     |____|      |____|        |____|      |____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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(PORT_PRIO_TYPE uXPriority_)
00041 {
00042     m_uXPriority = uXPriority_;
00043 }
00044
00045 //---------------------------------------------------------------------------
00046 void ThreadList::SetMapPointer(PriorityMap* pclMap_)
00047 {
00048     m_pclMap = pclMap_;
00049 }
00050
00051 //---------------------------------------------------------------------------
00052 void ThreadList::Add(LinkListNode* node_)
00053 {
00054     CircularLinkList::Add(node_);
00055     CircularLinkList::PivotForward();
00056
00057     // We've specified a bitmap for this threadlist
00058     if (m_pclMap) {
00059         // Set the flag for this priority level
00060         m_pclMap->Set(m_uXPriority);
00061     }
00062 }
00063
00064 //---------------------------------------------------------------------------
00065 void ThreadList::AddPriority(LinkListNode* node_)
00066 {
00067     Thread* pclCurr = static_cast<Thread*>(GetHead());
00068     if (!pclCurr) {
00069         Add(node_);
00070         return;
00071     }
00072     PORT_PRIO_TYPE uXHeadPri = pclCurr->GetCurPriority();
00073
00074     Thread* pclTail = static_cast<Thread*>(GetTail());
00075     Thread* pclNode = static_cast<Thread*>(node_);
00076
00077     // Set the threadlist's priority level, flag pointer, and then add the
00078     // thread to the threadlist
00079     PORT_PRIO_TYPE uXPriority = pclNode->GetCurPriority();
00080     do {
00081         if (uXPriority > pclCurr->GetCurPriority()) {
00082             break;
00083         }
00084         pclCurr = static_cast<Thread*>(pclCurr->GetNext());
00085     } while (pclCurr != pclTail);
00086
00087     // Insert pclNode before pclCurr in the linked list.
00088     InsertNodeBefore(pclNode, pclCurr);
00089
00090     // If the priority is greater than current head, reset
00091     // the head pointer.
00092     if (uXPriority > uXHeadPri) {
00093         m_pstHead = pclNode;
00094         m_pstTail = m_pstHead->prev;
00095     } else if (pclNode->GetNext() == m_pstHead) {
00096         m_pstTail = pclNode;
00097     }
00098 }
00099
00100 //---------------------------------------------------------------------------
00101 void ThreadList::Add(LinkListNode* node_, PriorityMap* pclMap_,
00102        PORT_PRIO_TYPE uXPriority_)
00102 {
00103     // Set the threadlist's priority level, flag pointer, and then add the
00104     // thread to the threadlist
00105     SetPriority(uXPriority_);
00106     SetMapPointer(pclMap_);
00107     Add(node_);
00108 }
```

```
00109
00110 //---------------------------------------------------------------------------
00111 void ThreadList::Remove(LinkListNode* node_)
00112 {
00113     // Remove the thread from the list
00114     CircularLinkList::Remove(node_);
00115
00116     // If the list is empty...
00117     if (!m_pstHead && m_pclMap) {
00118         // Clear the bit in the bitmap at this priority level
00119         m_pclMap->Clear(m_uXPriority);
00120     }
00121 }
00122
00123 //---------------------------------------------------------------------------
00124 Thread* ThreadList::HighestWaiter()
00125 {
00126     return static_cast<Thread*>(GetHead());
00127 }
```

20.117 /home/moslevin/mark3-source/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"
```

20.117.1 Detailed Description

Timer implementations.

Definition in file timer.cpp.

20.118 timer.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__    |__   _|__    |__   _|__    |__   _____
00004 |    \  /    |  |    \     |      |    ||  |/ /    ||___    |
00005 |     \/     |  ||    \         ||      ||      \      ||___     |
00006 |__/\__/|__||__|\__\   _||__|\__\   _||__|\__\   _||_____|
00007      |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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     if (m_u8Flags & TIMERLIST_FLAG_ACTIVE) {
00051         return;
00052     }
00053
00054     SetIntervalMSeconds(u32IntervalMs_);
00055     m_u32TimerTolerance = 0;
00056     m_pfCallback        = pfCallback_;
00057     m_pvData            = pvData_;
00058
00059     if (!bRepeat_) {
00060         m_u8Flags = TIMERLIST_FLAG_ONE_SHOT;
00061     } else {
00062         m_u8Flags = 0;
00063     }
00064
00065     Start();
00066 }
00067
00068 //---------------------------------------------------------------------------
00069 void Timer::Start(
00070     bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_,
      TimerCallback_t pfCallback_, void* pvData_)
00071 {
00072     if (m_u8Flags & TIMERLIST_FLAG_ACTIVE) {
00073         return;
00074     }
00075
00076     m_u32TimerTolerance = MSECONDS_TO_TICKS(u32ToleranceMs_);
00077     Start(bRepeat_, u32IntervalMs_, pfCallback_, pvData_);
00078 }
00079
00080 //---------------------------------------------------------------------------
00081 void Timer::Start()
00082 {
00083     if (m_u8Flags & TIMERLIST_FLAG_ACTIVE) {
00084         return;
00085     }
00086
00087     m_pclOwner = Scheduler::GetCurrentThread();
00088     TimerScheduler::Add(this);
00089 }
00090
00091 //---------------------------------------------------------------------------
00092 void Timer::Stop()
00093 {
00094     if (!(m_u8Flags & TIMERLIST_FLAG_ACTIVE)) {
00095         return;
00096     }
00097     TimerScheduler::Remove(this);
00098 }
00099
00100 //---------------------------------------------------------------------------
00101 void Timer::SetIntervalTicks(uint32_t u32Ticks_)
00102 {
00103     m_u32Interval = u32Ticks_;
00104 }
00105
00106 //---------------------------------------------------------------------------
00108 //---------------------------------------------------------------------------
00109 void Timer::SetIntervalSeconds(uint32_t u32Seconds_)
00110 {
00111     m_u32Interval = SECONDS_TO_TICKS(u32Seconds_);
00112 }
00113
00114 //---------------------------------------------------------------------------
00115 void Timer::SetIntervalMSeconds(uint32_t u32MSeconds_)
00116 {
00117     m_u32Interval = MSECONDS_TO_TICKS(u32MSeconds_);
00118 }
```

```
00119
00120 //---------------------------------------------------------------------
00121 void Timer::SetIntervalUSeconds(uint32_t u32USeconds_)
00122 {
00123 #if KERNEL_TIMERS_TICKLESS
00124     if (u32USeconds_ < KERNEL_TIMERS_MINIMUM_DELAY_US) {
00125         u32USeconds_ = KERNEL_TIMERS_MINIMUM_DELAY_US;
00126     }
00127 #endif
00128
00129     m_u32Interval = USECONDS_TO_TICKS(u32USeconds_);
00130 }
00131
00132 //---------------------------------------------------------------------
00133 void Timer::SetTolerance(uint32_t u32Ticks_)
00134 {
00135     m_u32TimerTolerance = u32Ticks_;
00136 }
00137
00138 #endif
```

20.119 /home/moslevin/mark3-source/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"
```

20.119.1 Detailed Description

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

Definition in file timerlist.cpp.

20.120 timerlist.cpp

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__    |__    __|__  |__    _____
00004 |    \  /    | ||    \       ||      |       || |/ /        ||___   |
00005 |     \/     | ||     \      ||      |       ||    \        ||__    |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
00007     |_____|       |_____|       |_____|       |_____|
00008
00009 --[Mark3 Realtime Platform]---------------------------------------------
00010
00011 Copyright (c) 2012-2016 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         bStart = 1;
00068     }
00069     if (pclListNode_->m_u32Interval < PORT_MIN_TIMER_TICKS) {
00070         pclListNode_->m_u32Interval = PORT_MIN_TIMER_TICKS;
00071     }
00072 #endif
00073
00074     pclListNode_->ClearNode();
00075     DoubleLinkList::Add(pclListNode_);
00076
00077     // Set the initial timer value
00078     pclListNode_->m_u32TimeLeft = pclListNode_->m_u32Interval;
00079
00080 #if KERNEL_TIMERS_TICKLESS
00081     if (!bStart) {
00082         // If the new interval is less than the amount of time remaining...
00083         lDelta = (int32_t)((uint32_t)KernelTimer::TimeToExpiry() - pclListNode_->
    m_u32Interval);
00084
00085         if (lDelta > 0) {
00086             // Set the new expiry time on the timer.
00087             m_u32NextWakeup = (uint32_t)
    KernelTimer::SubtractExpiry((uint32_t)lDelta);
00088         }
00089     } else {
00090         m_u32NextWakeup = pclListNode_->m_u32Interval;
00091         KernelTimer::SetExpiry(m_u32NextWakeup);
00092         KernelTimer::Start();
00093     }
00094 #endif
00095
00096     // Set the timer as active.
00097     pclListNode_->m_u8Flags |= TIMERLIST_FLAG_ACTIVE;
00098     CS_EXIT();
00099 }
00100
00101 //---------------------------------------------------------------------------
00102 void TimerList::Remove(Timer* pclLinkListNode_)
00103 {
00104     CS_ENTER();
00105
00106     DoubleLinkList::Remove(pclLinkListNode_);
00107     pclLinkListNode_->m_u8Flags &= ~TIMERLIST_FLAG_ACTIVE;
00108
00109 #if KERNEL_TIMERS_TICKLESS
00110     if (this->GetHead() == NULL) {
00111         KernelTimer::Stop();
00112     }
00113 #endif
00114
00115     CS_EXIT();
00116 }
00117
00118 //--------------------------------------------------------------------------
00119 void TimerList::Process(void)
00120 {
```

```
00121 #if KERNEL_TIMERS_TICKLESS
00122     uint32_t u32NewExpiry;
00123     uint32_t u32Overtime;
00124     bool     bContinue;
00125 #endif
00126
00127     Timer* pclNode;
00128     Timer* pclPrev;
00129
00130 #if KERNEL_USE_QUANTUM
00131     Quantum::SetInTimer();
00132 #endif
00133 #if KERNEL_TIMERS_TICKLESS
00134     // Clear the timer and its expiry time - keep it running though
00135     KernelTimer::ClearExpiry();
00136     do {
00137 #endif
00138         pclNode = static_cast<Timer*>(GetHead());
00139         pclPrev = NULL;
00140
00141 #if KERNEL_TIMERS_TICKLESS
00142         bContinue   = 0;
00143         u32NewExpiry = MAX_TIMER_TICKS; // Used to indicate that no timers are pending
00144 #endif
00145
00146         // Subtract the elapsed time interval from each active timer.
00147         while (pclNode) {
00148             // Active timers only...
00149             if (pclNode->m_u8Flags & TIMERLIST_FLAG_ACTIVE) {
00150 // Did the timer expire?
00151 #if KERNEL_TIMERS_TICKLESS
00152                 if (pclNode->m_u32TimeLeft <= m_u32NextWakeup)
00153 #else
00154                 pclNode->m_u32TimeLeft--;
00155                 if (0 == pclNode->m_u32TimeLeft)
00156 #endif
00157                 {
00158                     // Yes - set the "callback" flag - we'll execute the callbacks later
00159                     pclNode->m_u8Flags |= TIMERLIST_FLAG_CALLBACK;
00160
00161                     if (pclNode->m_u8Flags & TIMERLIST_FLAG_ONE_SHOT) {
00162                         // If this was a one-shot timer, deactivate the timer.
00163                         pclNode->m_u8Flags |= TIMERLIST_FLAG_EXPIRED;
00164                         pclNode->m_u8Flags &= ~TIMERLIST_FLAG_ACTIVE;
00165                     } else {
00166                         // Reset the interval timer.
00168                         // I think we're good though...
00169                         pclNode->m_u32TimeLeft = pclNode->m_u32Interval;
00170
00171 #if KERNEL_TIMERS_TICKLESS
00172                         // If the time remaining (plus the length of the tolerance interval)
00173                         // is less than the next expiry interval, set the next expiry interval.
00174                         uint32_t u32Tmp = pclNode->m_u32TimeLeft + pclNode->m_u32TimerTolerance;
00175
00176                         if (u32Tmp < u32NewExpiry) {
00177                             u32NewExpiry = u32Tmp;
00178                         }
00179 #endif
00180                     }
00181                 }
00182 #if KERNEL_TIMERS_TICKLESS
00183                 else {
00184                     // Not expiring, but determine how long to run the next timer interval for.
00185                     pclNode->m_u32TimeLeft -= m_u32NextWakeup;
00186                     if (pclNode->m_u32TimeLeft < u32NewExpiry) {
00187                         u32NewExpiry = pclNode->m_u32TimeLeft;
00188                     }
00189                 }
00190 #endif
00191             }
00192             pclNode = static_cast<Timer*>(pclNode->GetNext());
00193         }
00194
00195         // Process the expired timers callbacks.
00196         pclNode = static_cast<Timer*>(GetHead());
00197         while (pclNode) {
00198             pclPrev = pclNode;
00199             pclNode = static_cast<Timer*>(pclNode->GetNext());
00200
00201             // If the timer expired, run the callbacks now.
00202             if (pclPrev->m_u8Flags & TIMERLIST_FLAG_CALLBACK) {
00203                 bool bRemove = false;
00204                 // If this was a one-shot timer, tag it for removal
00205                 if (pclPrev->m_u8Flags & TIMERLIST_FLAG_ONE_SHOT) {
00206                     bRemove = true;
00207                 }
00208
```

```
00209                   // Run the callback. these callbacks must be very fast...
00210                   pclPrev->m_pfCallback(pclPrev->m_pclOwner, pclPrev->m_pvData);
00211                   pclPrev->m_u8Flags &= ~TIMERLIST_FLAG_CALLBACK;
00212
00213                   // Remove one-shot-timers
00214                   if (bRemove) {
00215                       Remove(pclPrev);
00216                   }
00217               }
00218           }
00219
00220 #if KERNEL_TIMERS_TICKLESS
00221           // Check to see how much time has elapsed since the time we
00222           // acknowledged the interrupt...
00223           u32Overtime = (uint32_t)KernelTimer::GetOvertime();
00224
00225           if (u32Overtime >= u32NewExpiry) {
00226               m_u32NextWakeup = u32Overtime;
00227               bContinue       = 1;
00228           }
00229
00230           // If it's taken longer to go through this loop than would take us to
00231           // the next expiry, re-run the timing loop
00232
00233       } while (bContinue);
00234
00235       // This timer elapsed, but there's nothing more to do...
00236       // Turn the timer off.
00237       if (u32NewExpiry >= MAX_TIMER_TICKS) {
00238           KernelTimer::Stop();
00239       } else {
00240           // Update the timer with the new "Next Wakeup" value, plus whatever
00241           // overtime has accumulated since the last time we called this handler
00242
00243           m_u32NextWakeup = (uint32_t)KernelTimer::SetExpiry(
00244       u32NewExpiry + u32Overtime);
00244       }
00245 #endif
00246 #if KERNEL_USE_QUANTUM
00247       Quantum::ClearInTimer();
00248 #endif
00249 }
00250
00251 #endif // KERNEL_USE_TIMERS
```

20.121 /home/moslevin/mark3-source/embedded/kernel/tracebuffer.cpp File Reference

Kernel trace buffer class definition.

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

20.121.1 Detailed Description

Kernel trace buffer class definition.

Definition in file tracebuffer.cpp.

20.122 tracebuffer.cpp

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

```
00008
00009 --[Mark3 Realtime Platform]-----------------------------------------------------
00010
00011 Copyright (c) 2012-2016 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
00023 #define _CAN_HAS_DEBUG
00024 //--[Autogenerated - Do Not Modify]------------------------------------------------
00025 #include "dbg_file_list.h"
00026 #include "buffalogger.h"
00027 #if defined(DBG_FILE)
00028 #error "Debug logging file token already defined!  Bailing."
00029 #else
00030 #define DBG_FILE _DBG___KERNEL_TRACEBUFFER_CPP
00031 #endif
00032
00033 #include "kerneldebug.h"
00034
00035 //--[End Autogenerated content]----------------------------------------------------
00036
00037 #if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
00038 //---------------------------------------------------------------------------------
00039 TraceBufferCallback_t  TraceBuffer::m_pfCallback;
00040 uint16_t               TraceBuffer::m_u16Index;
00041 uint16_t               TraceBuffer::m_u16SyncNumber;
00042 uint16_t               TraceBuffer::m_au16Buffer[(TRACE_BUFFER_SIZE / sizeof(uint16_t))];
00043
00044 //---------------------------------------------------------------------------------
00045 void TraceBuffer::Init()
00046 {
00047 }
00048
00049 //---------------------------------------------------------------------------------
00050 void TraceBuffer::Write(uint16_t* pu16Data_, uint16_t u16Size_)
00051 {
00052     // Pipe the data directly to the circular buffer
00053     uint16_t u16Start;
00054
00055     // Update the circular buffer index in a critical section. The
00056     // rest of the operations can take place in any context.
00057     CS_ENTER();
00058     uint16_t u16NextIndex;
00059     u16Start     = m_u16Index;
00060     u16NextIndex = m_u16Index + u16Size_;
00061     if (u16NextIndex >= (sizeof(m_au16Buffer) / sizeof(uint16_t))) {
00062         u16NextIndex -= (sizeof(m_au16Buffer) / sizeof(uint16_t));
00063     }
00064     m_u16Index = u16NextIndex;
00065     CS_EXIT();
00066
00067     // Write the data into the circular buffer.
00068     uint16_t i;
00069     bool    bCallback = false;
00070     bool    bPingPong = false;
00071     for (i = 0; i < u16Size_; i++) {
00072         m_au16Buffer[u16Start++] = pu16Data_[i];
00073         if (u16Start >= (sizeof(m_au16Buffer) / sizeof(uint16_t))) {
00074             u16Start = 0;
00075             bCallback = true;
00076         } else if (u16Start == ((sizeof(m_au16Buffer) / sizeof(uint16_t)) / 2)) {
00077             bPingPong = true;
00078             bCallback = true;
00079         }
00080     }
00081
00082     // Done writing - see if there's a 50% or rollover callback
00083     if (bCallback && m_pfCallback) {
00084         uint16_t u16Size = (sizeof(m_au16Buffer) / sizeof(uint16_t)) / 2;
00085         if (bPingPong) {
00086             m_pfCallback(m_au16Buffer, u16Size, bPingPong);
00087         } else {
00088             m_pfCallback(m_au16Buffer + u16Size, u16Size, bPingPong);
00089         }
00090     }
00091 }
00092
00093 #endif
```

20.123 /home/moslevin/mark3-source/embedded/libs/mark3c/public/fake_types.h File Reference

C-struct definitions that mirror.

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

20.123.1 Detailed Description

C-struct definitions that mirror.

This header contains a set of "fake" structures that have the same memory layout as the kernel objects in C++ (taking into account inheritence, etc.). These are used for sizing the opaque data blobs that are declared in C, which then become instantiated as C++ kernel objects via the bindings provided.

Definition in file fake_types.h.

20.124 fake_types.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003   ___|    _|__  __|_ |__  __|_ |__  __|_ |__  _____
00004  |    \  /  | ||    |        ||        ||    |/ /    ||___ |
00005  |     \/   | ||     \       ||     \       ||     \       ||__   |
00006  |__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||_____|
00007       |_____|        |_____|        |_____|        |_____|
00008
00009 --[Mark3 Realtime Platform]------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00026 #include "kerneltypes.h"
00027 #include "mark3cfg.h"
00028
00029 #ifndef __FAKE_TYPES_H__
00030 #define __FAKE_TYPES_H__
00031
00032 #if defined(__cplusplus)
00033 extern "C" {
00034 #endif
00035
00036 //------------------------------------------------------------------------
00037 typedef struct {
00038     void* prev;
00039     void* next;
00040 } Fake_LinkedListNode;
00041
00042 //------------------------------------------------------------------------
00043 typedef struct {
00044     void* vtab_ptr;
00045     void* head;
00046     void* tail;
00047 } Fake_LinkedList;
00048
00049 //------------------------------------------------------------------------
00050 typedef struct {
00051     Fake_LinkedList fake_list;
00052     PORT_PRIO_TYPE       m_uXPriority;
00053     void*            m_pclMap;
00054 } Fake_ThreadList;
00055
00056 //------------------------------------------------------------------------
00057 typedef struct {
00058     Fake_LinkedListNode m_ll_node;
00059     uint8_t             m_u8Flags;
00060     void*               m_pfCallback;
00061     uint32_t            m_u32Interval;
00062     uint32_t            m_u32TimeLeft;
00063     uint32_t            m_u32TimerTolerance;
00064     void*               m_pclOwner;
00065     void*               m_pvData;
```

```
00066 } Fake_Timer;
00067
00068 //-------------------------------------------------------------------------
00069 typedef struct {
00070     Fake_LinkedListNode m_ll_node;
00071     K_WORD*              m_pwStackTop;
00072     K_WORD*              m_pwStack;
00073     uint8_t              m_u8ThreadID;
00074     PORT_PRIO_TYPE          m_uXPriority;
00075     PORT_PRIO_TYPE          m_uXCurPriority;
00076     uint8_t              m_eState;
00077 #if KERNEL_USE_THREADNAME
00078     const char* m_szName;
00079 #endif
00080     uint16_t m_u16StackSize;
00081     void*    m_pclCurrent;
00082     void*    m_pclOwner;
00083     void*    m_pfEntryPoint;
00084     void*    m_pvArg;
00085 #if KERNEL_USE_QUANTUM
00086     uint16_t m_u16Quantum;
00087 #endif
00088 #if KERNEL_USE_EVENTFLAG
00089     uint16_t m_u16FlagMask;
00090     uint8_t  m_eFlagMode;
00091 #endif
00092 #if KERNEL_USE_TIMEOUTS || KERNEL_USE_SLEEP
00093     Fake_Timer m_clTimer;
00094 #endif
00095 #if KERNEL_USE_TIMEOUTS
00096     bool m_bExpired;
00097 #endif
00098 } Fake_Thread;
00099
00100 //-------------------------------------------------------------------------
00101 typedef struct {
00102     Fake_ThreadList thread_list;
00103     uint16_t        m_u16Value;
00104     uint16_t        m_u16MaxValue;
00105 } Fake_Semaphore;
00106
00107 //-------------------------------------------------------------------------
00108 typedef struct {
00109     Fake_ThreadList thread_list;
00110     uint8_t         m_u8Recurse;
00111     bool            m_bReady;
00112     uint8_t         m_u8MaxPri;
00113     void*           m_pclOwner;
00114 } Fake_Mutex;
00115
00116 //-------------------------------------------------------------------------
00117 typedef struct {
00118     Fake_LinkedListNode list_node;
00119     void*               m_pvData;
00120     uint16_t            m_u16Code;
00121 } Fake_Message;
00122
00123 //-------------------------------------------------------------------------
00124 typedef struct {
00125     Fake_Semaphore  m_clSemaphore;
00126     Fake_LinkedList m_clLinkList;
00127 } Fake_MessageQueue;
00128
00129 //-------------------------------------------------------------------------
00130 typedef struct {
00131     uint16_t        m_u16Head;
00132     uint16_t        m_u16Tail;
00133     uint16_t        m_u16Count;
00134     uint16_t        m_u16Free;
00135     uint16_t        m_u16ElementSize;
00136     void*           m_pvBuffer;
00137     Fake_Semaphore m_clRecvSem;
00138 #if KERNEL_USE_TIMEOUTS
00139     Fake_Semaphore m_clSendSem;
00140 #endif
00141 } Fake_Mailbox;
00142
00143 //-------------------------------------------------------------------------
00144 typedef struct {
00145     Fake_ThreadList thread_list;
00146 } Fake_Notify;
00147
00148 //-------------------------------------------------------------------------
00149 typedef struct {
00150     Fake_ThreadList thread_list;
00151     uint16_t        m_u16EventFlag;
00152 } Fake_EventFlag;
```

```
00153
00154 #if defined(__cplusplus)
00155 }
00156 #endif
00157
00158 #endif // __FAKE_TYPES_H__
```

20.125 /home/moslevin/mark3-source/embedded/libs/mark3c/public/mark3c.h File Reference

Header providing C-language API bindings for the Mark3 kernel.

```
#include "mark3cfg.h"
#include "kerneltypes.h"
#include "fake_types.h"
#include "driver3c.h"
#include <stdint.h>
#include <stdbool.h>
```

Typedefs

- typedef void ∗ EventFlag_t

    EventFlag opaque handle data type.
- typedef void ∗ Mailbox_t

    Mailbox opaque handle data type.
- typedef void ∗ Message_t

    Message opaque handle data type.
- typedef void ∗ MessageQueue_t

    MessageQueue opaque handle data type.
- typedef void ∗ Mutex_t

    Mutex opaque handle data type.
- typedef void ∗ Notify_t

    Notification object opaque handle data type.
- typedef void ∗ Semaphore_t

    Semaphore opaque handle data type.
- typedef void ∗ Thread_t

    Thread opaque handle data type.
- typedef void ∗ Timer_t

    Timer opaque handle data type.

Functions

- void ∗ AutoAlloc (uint16_t u16Size_)

    AutoAlloc.
- Semaphore_t Alloc_Semaphore (void)

    Alloc_Semaphore.
- Mutex_t Alloc_Mutex (void)

    Alloc_Mutex.
- EventFlag_t Alloc_EventFlag (void)

    Alloc_EventFlag.
- Message_t Alloc_Message (void)

Alloc_Message.

- MessageQueue_t Alloc_MessageQueue (void)

    Alloc_MessageQueue.

- Notify_t Alloc_Notify (void)

    Alloc_Notify.

- Mailbox_t Alloc_Mailbox (void)

    Alloc_Mailbox.

- Thread_t Alloc_Thread (void)

    Alloc_Thread.

- Timer_t Alloc_Timer (void)

    Alloc_Timer.

- void Kernel_Init (void)

    Kernel_Init.

- void Kernel_Start (void)

    Kernel_Start.

- bool Kernel_IsStarted (void)

    Kernel_IsStarted.

- void Kernel_SetPanic (PanicFunc_t pfPanic_)

    Kernel_SetPanic.

- bool Kernel_IsPanic (void)

    Kernel_IsPanic.

- void Kernel_Panic (uint16_t u16Cause_)

    Kernel_Panic.

- void Kernel_SetIdleFunc (IdleFunc_t pfIdle_)

    Kernel_SetIdleFunc.

- void Kernel_SetThreadCreateCallout (thread_create_callout_t pfCreate_)

    Kernel_SetThreadCreateCallout.

- void Kernel_SetThreadExitCallout (thread_exit_callout_t pfExit_)

    Kernel_SetThreadExitCallout.

- void Kernel_SetThreadContextSwitchCallout (thread_context_callout_t pfContext_)

    Kernel_SetThreadContextSwitchCallout.

- thread_create_callout_t Kernel_GetThreadCreateCallout (void)

    Kernel_GetThreadCreateCallout.

- thread_exit_callout_t Kernel_GetThreadExitCallout (void)

    Kernel_GetThreadExitCallout.

- thread_context_callout_t Kernel_GetThreadContextSwitchCallout (void)

    Kernel_GetThreadContextSwitchCallout.

- void Scheduler_Enable (bool bEnable_)

    Scheduler_Enable.

- bool Scheduler_IsEnabled (void)

    Scheduler_IsEnabled.

- Thread_t Scheduler_GetCurrentThread (void)

    Scheduler_GetCurrentThread.

- void Thread_Init (Thread_t handle, K_WORD *pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE u↩
XPriority_, ThreadEntry_t pfEntryPoint_, void *pvArg_)

    Thread_Init.

- void Thread_Start (Thread_t handle)

    Thread_Start.

- void Thread_Stop (Thread_t handle)

    Thread_Stop.

- PORT_PRIO_TYPE Thread_GetPriority (Thread_t handle)

Thread_GetPriority.

- PORT_PRIO_TYPE Thread_GetCurPriority (Thread_t handle)

   Thread_GetCurPriority.

- void Thread_SetQuantum (Thread_t handle, uint16_t u16Quantum_)

   Thread_SetQuantum.

- uint16_t Thread_GetQuantum (Thread_t handle)

   Thread_GetQuantum.

- void Thread_SetPriority (Thread_t handle, PORT_PRIO_TYPE uXPriority_)

   Thread_SetPriority.

- void Thread_Exit (Thread_t handle)

   Thread_Exit.

- void Thread_Sleep (uint32_t u32TimeMs_)

   Thread_Sleep.

- void Thread_USleep (uint32_t u32TimeUs_)

   Thread_USleep.

- void Thread_Yield (void)

   Thread_Yield.

- void Thread_SetID (Thread_t handle, uint8_t u8ID_)

   Thread_SetID.

- uint8_t Thread_GetID (Thread_t handle)

   Thread_GetID.

- uint16_t Thread_GetStackSlack (Thread_t handle)

   Thread_GetStackSlack.

- ThreadState_t Thread_GetState (Thread_t handle)

   Thread_GetState.

- void Timer_Init (Timer_t handle)

   Timer_Init.

- void Timer_Start (Timer_t handle, bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_↩
   , TimerCallbackC_t pfCallback_, void *pvData_)

   Timer_Start.

- void Timer_Restart (Timer_t handle)

   Timer_Restart.

- void Timer_Stop (Timer_t handle)

   Timer_Stop.

- void Semaphore_Init (Semaphore_t handle, uint16_t u16InitVal_, uint16_t u16MaxVal_)

   Semaphore_Init.

- void Semaphore_Post (Semaphore_t handle)

   Semaphore_Post.

- void Semaphore_Pend (Semaphore_t handle)

   Semaphore_Pend.

- bool Semaphore_TimedPend (Semaphore_t handle, uint32_t u32WaitTimeMS_)

   Semaphore_TimedPend.

- void Mutex_Init (Mutex_t handle)

   Mutex_Init.

- void Mutex_Claim (Mutex_t handle)

   Mutex_Claim.

- void Mutex_Release (Mutex_t handle)

   Mutex_Release.

- bool Mutex_TimedClaim (Mutex_t handle, uint32_t u32WaitTimeMS_)

   Mutex_TimedClaim.

- void EventFlag_Init (EventFlag_t handle)

EventFlag_Init.

- uint16_t EventFlag_Wait (EventFlag_t handle, uint16_t u16Mask_, EventFlagOperation_t eMode_)

  EventFlag_Wait.

- uint16_t EventFlag_TimedWait (EventFlag_t handle, uint16_t u16Mask_, EventFlagOperation_t eMode_↩
  , uint32_t u32TimeMS_)

  EventFlag_TimedWait.

- void EventFlag_Set (EventFlag_t handle, uint16_t u16Mask_)

  EventFlag_Set.

- void EventFlag_Clear (EventFlag_t handle, uint16_t u16Mask_)

  EventFlag_Clear.

- uint16_t EventFlag_GetMask (EventFlag_t handle)

  EventFlag_GetMask.

- void Notify_Init (Notify_t handle)

  Notify_Init.

- void Notify_Signal (Notify_t handle)

  Notify_Signal.

- void Notify_Wait (Notify_t handle, bool ∗pbFlag_)

  Notify_Wait.

- bool Notify_TimedWait (Notify_t handle, uint32_t u32WaitTimeMS_, bool ∗pbFlag_)

  Notify_TimedWait.

- void Message_Init (Message_t handle)

  Message_Init.

- void Message_SetData (Message_t handle, void ∗pvData_)

  Message_SetData.

- void ∗ Message_GetData (Message_t handle)

  Message_GetData.

- void Message_SetCode (Message_t handle, uint16_t u16Code_)

  Message_SetCode.

- uint16_t Message_GetCode (Message_t handle)

  Message_GetCode.

- void GlobalMessagePool_Push (Message_t handle)

  GlobalMessagePool_Push.

- Message_t GlobalMessagePool_Pop (void)

  GlobalMessagePool_Pop.

- void MessageQueue_Init (MessageQueue_t handle)

  MessageQueue_Init.

- Message_t MessageQueue_Receive (MessageQueue_t handle)

  MessageQueue_Receive.

- Message_t MessageQueue_TimedReceive (MessageQueue_t handle, uint32_t u32TimeWaitMS_)

  MessageQueue_TimedReceive.

- void MessageQueue_Send (MessageQueue_t handle, Message_t hMessage_)

  MessageQueue_Send.

- uint16_t MessageQueue_GetCount (void)

  MessageQueue_GetCount.

- void Mailbox_Init (Mailbox_t handle, void ∗pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_)

  Mailbox_Init.

- bool Mailbox_Send (Mailbox_t handle, void ∗pvData_)

  Mailbox_Send.

- bool Mailbox_SendTail (Mailbox_t handle, void ∗pvData_)

  Mailbox_SendTail.

- bool Mailbox_TimedSend (Mailbox_t handle, void ∗pvData_, uint32_t u32TimeoutMS_)

Mailbox_TimedSend.

- bool Mailbox_TimedSendTail (Mailbox_t handle, void ∗pvData_, uint32_t u32TimeoutMS_)

    Mailbox_TimedSendTail.

- void Mailbox_Receive (Mailbox_t handle, void ∗pvData_)

    Mailbox_Receive.

- void Mailbox_ReceiveTail (Mailbox_t handle, void ∗pvData_)

    Mailbox_ReceiveTail.

- bool Mailbox_TimedReceive (Mailbox_t handle, void ∗pvData_, uint32_t u32TimeoutMS_)

    Mailbox_TimedReceive.

- bool Mailbox_TimedReceiveTail (Mailbox_t handle, void ∗pvData_, uint32_t u32TimeoutMS_)

    Mailbox_TimedReceiveTail.

- uint16_t Mailbox_GetFreeSlots (Mailbox_t handle)

    Mailbox_GetFreeSlots.

- bool Mailbox_IsFull (Mailbox_t handle)

    Mailbox_IsFull.

- bool Mailbox_IsEmpty (Mailbox_t handle)

    Mailbox_IsEmpty.

- void KernelAware_ProfileInit (const char ∗szStr_)

    KernelAware_ProfileInit.

- void KernelAware_ProfileStart (void)

    KernelAware_ProfileStart.

- void KernelAware_ProfileStop (void)

    KernelAware_ProfileStop.

- void KernelAware_ProfileReport (void)

    KernelAware_ProfileReport.

- void KernelAware_ExitSimulator (void)

    KernelAware_ExitSimulator.

- void KernelAware_Print (const char ∗szStr_)

    KernelAware_Print.

- void KernelAware_Trace (uint16_t u16File_, uint16_t u16Line_)

    KernelAware_Trace.

- void KernelAware_Trace1 (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_)

    KernelAware_Trace1.

- void KernelAware_Trace2 (uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_)

    KernelAware_Trace2.

- bool KernelAware_IsSimulatorAware (void)

    KernelAware_IsSimulatorAware.

20.125.1 Detailed Description

Header providing C-language API bindings for the Mark3 kernel.

Definition in file mark3c.h.

20.125.2 Function Documentation

20.125.2.1 EventFlag_t Alloc_EventFlag (void)

Alloc_EventFlag.

See also

EventFlag∗ AutoAlloc::NewEventFlag()

Returns

Handle to an allocated object, or NULL if heap exhausted

20.125.2.2 Mailbox_t Alloc_Mailbox (void)

Alloc_Mailbox.

See also

Mailbox∗ AutoAlloc::NewMailbox()

Returns

Handle to an allocated object, or NULL if heap exhausted

20.125.2.3 Message_t Alloc_Message (void)

Alloc_Message.

See also

AutoAlloc::NewMessage()

Returns

Handle to an allocated object, or NULL if heap exhausted

20.125.2.4 MessageQueue_t Alloc_MessageQueue (void)

Alloc_MessageQueue.

See also

MesageQueue∗ AutoAlloc::NewMessageQueue()

Returns

Handle to an allocated object, or NULL if heap exhausted

20.125.2.5 Mutex_t Alloc_Mutex (void)

Alloc_Mutex.

See also

Mutex∗ AutoAlloc::NewMutex()

Returns

Handle to an allocated object, or NULL if heap exhausted

20.125.2.6   Notify_t Alloc_Notify (void)

Alloc_Notify.

See also

> Notify∗ AutoAlloc::NewNotify()

Returns

> Handle to an allocated object, or NULL if heap exhausted

20.125.2.7   Semaphore_t Alloc_Semaphore (void)

Alloc_Semaphore.

See also

> Semaphore∗ AutoAlloc::NewSemaphore()

Returns

> Handle to an allocated object, or NULL if heap exhausted

20.125.2.8   Thread_t Alloc_Thread (void)

Alloc_Thread.

See also

> Thread∗ AutoAlloc::NewThread()

Returns

> Handle to an allocated object, or NULL if heap exhausted

20.125.2.9   Timer_t Alloc_Timer (void)

Alloc_Timer.

See also

> Timer∗ AutoAlloc::NewTimer()

Returns

> Handle to an allocated object, or NULL if heap exhausted

20.125.2.10   void∗ AutoAlloc (uint16_t *u16Size_*)

AutoAlloc.

See also

> void∗ AutoAlloc::Allocate(uint16_t u16Size_)

Parameters

*u16Size_*	Size in bytes to allocate from the one-time-allocate heap

Returns

> Pointer to an allocated blob of memory, or NULL if heap exhausted

20.125.2.11 void EventFlag_Clear (EventFlag_t *handle,* uint16_t *u16Mask_*)

EventFlag_Clear.

See also

> void EventFlag::Clear(uint16_t u16Mask_)

Parameters

handle	Handle of the event flag object
*u16Mask_*	Bits to clear in the eventflag's internal condition regster

20.125.2.12 uint16_t EventFlag_GetMask (EventFlag_t *handle*)

EventFlag_GetMask.

See also

> void EventFlag::GetMask()

Parameters

handle	Handle of the event flag object

Returns

> Return the current bitmask

20.125.2.13 void EventFlag_Init (EventFlag_t *handle*)

EventFlag_Init.

See also

> void EventFlag::Init()

Parameters

handle	Handle of the event flag object

20.125.2.14 void EventFlag_Set (EventFlag_t *handle,* uint16_t *u16Mask_*)

EventFlag_Set.

See also

> void EventFlag::Set(uint16_t u16Mask_)

Parameters

handle	Handle of the event flag object
*u16Mask_*	Bits to set in the eventflag's internal condition register

20.125.2.15 **uint16_t EventFlag_TimedWait (EventFlag_t** *handle,* **uint16_t** *u16Mask_,* **EventFlagOperation_t** *eMode_,* **uint32_t** *u32TimeMS_* **)**

EventFlag_TimedWait.

See also

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

Parameters

handle	Handle of the event flag object
*u16Mask_*	condition flags to wait for
*eMode_*	Specify conditions under which the thread will be unblocked
*u32TimeMS_*	Time in ms to wait before aborting the operation

Returns

bitfield contained in the eventflag on unblock, or 0 on expiry.

20.125.2.16 **uint16_t EventFlag_Wait (EventFlag_t** *handle,* **uint16_t** *u16Mask_,* **EventFlagOperation_t** *eMode_* **)**

EventFlag_Wait.

See also

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

Parameters

handle	Handle of the event flag object
*u16Mask_*	condition flags to wait for
*eMode_*	Specify conditions under which the thread will be unblocked

Returns

bitfield contained in the eventflag on unblock

20.125.2.17 **Message_t GlobalMessagePool_Pop (void)**

GlobalMessagePool_Pop.

See also

Message_t GlobalMessagePool::Pop()

Returns

Pointer to a Message object

20.125.2.18 void GlobalMessagePool_Push (Message_t *handle* **)**

GlobalMessagePool_Push.

See also

void GlobalMessagePool::Push()

Parameters

handle	Handle of the message object

20.125.2.19 thread_context_callout_t Kernel_GetThreadContextSwitchCallout (void)

Kernel_GetThreadContextSwitchCallout.

See also

Kernel::GetThreadContextSwitchCallout

Returns

Current function called on each context switch

20.125.2.20 thread_create_callout_t Kernel_GetThreadCreateCallout (void)

Kernel_GetThreadCreateCallout.

See also

Kernel::GetThreadCreateCallout

Returns

Current function called on each thread creation

20.125.2.21 thread_exit_callout_t Kernel_GetThreadExitCallout (void)

Kernel_GetThreadExitCallout.

See also

Kernel::GetThreadExitCallout

Returns

Current function called on each thread exit

20.125.2.22 void Kernel_Init (void)

Kernel_Init.

See also

void Kernel::Init()

20.125.2.23   bool Kernel_IsPanic (void)

Kernel_IsPanic.

See also

bool Kernel::IsPanic()

Returns

Whether or not the kernel is in a panic state

20.125.2.24   bool Kernel_IsStarted (void)

Kernel_IsStarted.

See also

bool Kernel::IsStarted()

Returns

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

20.125.2.25   void Kernel_Panic (uint16_t *u16Cause_*)

Kernel_Panic.

See also

void Kernel::Panic(uint16_t u16Cause_)

Parameters

*u16Cause_*	Reason for the kernel panic

20.125.2.26   void Kernel_SetIdleFunc (IdleFunc_t *pfIdle_*)

Kernel_SetIdleFunc.

See also

void Kernel::SetIdleFunc(IdleFunc_t pfIdle_)

Parameters

*pfIdle_*	Pointer to the idle function

20.125.2.27   void Kernel_SetPanic (PanicFunc_t *pfPanic_*)

Kernel_SetPanic.

See also

void Kernel::SetPanic(PanicFunc_t pfPanic_)

Parameters

*pfPanic_*	Panic function pointer

20.125.2.28   void Kernel_SetThreadContextSwitchCallout (thread_context_callout_t *pfContext_*)

Kernel_SetThreadContextSwitchCallout.

See also

> Kernel::SetThreadContextSwitchCallout

Parameters

*pfContext_*	Function to call prior to each context switch

20.125.2.29   void Kernel_SetThreadCreateCallout (thread_create_callout_t *pfCreate_*)

Kernel_SetThreadCreateCallout.

See also

> Kernel::SetThreadCreateCallout

Parameters

*pfCreate_*	Function to calll on thread creation

20.125.2.30   void Kernel_SetThreadExitCallout (thread_exit_callout_t *pfExit_*)

Kernel_SetThreadExitCallout.

See also

> Kernel::SetThreadExitCallout

Parameters

*pfExit_*	Function to call on thread exit

20.125.2.31   void Kernel_Start (void)

Kernel_Start.

See also

> void Kernel::Start()

20.125.2.32   void KernelAware_ExitSimulator (void)

KernelAware_ExitSimulator.

See also

> void KernelAware::ExitSimulator()

20.125.2.33 bool KernelAware_IsSimulatorAware (void)

KernelAware_IsSimulatorAware.

See also

void Kernel::IsSimulatorAware()

Returns

true if the runtime environment/simulator is aware that it is running the Mark3 kernel.

20.125.2.34 void KernelAware_Print (const char ∗ *szStr_*)

KernelAware_Print.

See also

void KernelAware::Print(const char ∗szStr_)

Parameters

*szStr_*	String to print to the kernel-aware simulator

20.125.2.35 void KernelAware_ProfileInit (const char ∗ *szStr_*)

KernelAware_ProfileInit.

See also

void KernelAware::ProfileInit(const char ∗szStr_);

Parameters

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

20.125.2.36 void KernelAware_ProfileReport (void)

KernelAware_ProfileReport.

See also

void KernelAware::ProfileReport()

20.125.2.37 void KernelAware_ProfileStart (void)

KernelAware_ProfileStart.

See also

void KernelAware::ProfileStart()

20.125.2.38   void KernelAware_ProfileStop (void)

KernelAware_ProfileStop.

See also

void KernelAware::ProfileStop()

20.125.2.39   void KernelAware_Trace (uint16_t *u16File_,* uint16_t *u16Line_*)

KernelAware_Trace.

See also

void KernelAware::Trace(uint16_t u16File_, uint16_t u16Line_);

Parameters

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

20.125.2.40   void KernelAware_Trace1 (uint16_t *u16File_,* uint16_t *u16Line_,* uint16_t *u16Arg1_*)

KernelAware_Trace1.

See also

void KernelAware::Trace(uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_);

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.

20.125.2.41   void KernelAware_Trace2 (uint16_t *u16File_,* uint16_t *u16Line_,* uint16_t *u16Arg1_,* uint16_t *u16Arg2_*)

KernelAware_Trace2.

See also

void KernelAware::Trace(uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t u16Arg2_);

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.

20.125.2.42   uint16_t Mailbox_GetFreeSlots (Mailbox_t *handle*)

Mailbox_GetFreeSlots.

See also

uint16_t Mailbox::GetFreeSlots()

Parameters

handle	Handle of the mailbox object

Returns

Number of free slots in the mailbox

20.125.2.43   void Mailbox_Init (Mailbox_t *handle,* **void** ∗ *pvBuffer_,* **uint16_t** *u16BufferSize_,* **uint16_t** *u16ElementSize_* **)**

Mailbox_Init.

See also

void Mailbox::Init(void ∗pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_)

Parameters

handle	Handle of the mailbox object
*pvBuffer_*	Pointer to the static buffer to use for the mailbox
u16BufferSize↩ *_*	Size of the mailbox buffer, in bytes
u16Element↩ *Size_*	Size of each envelope, in bytes

20.125.2.44   bool Mailbox_IsEmpty (Mailbox_t *handle* **)**

Mailbox_IsEmpty.

See also

bool Mailbox::IsEmpty()

Parameters

handle	Handle of the mailbox object

Returns

true if the mailbox is empty, false otherwise

20.125.2.45   bool Mailbox_IsFull (Mailbox_t *handle* **)**

Mailbox_IsFull.

See also

bool Mailbox::IsFull()

Parameters

handle	Handle of the mailbox object

Returns

true if the mailbox is full, false otherwise

20.125.2.46   void Mailbox_Receive (Mailbox_t *handle,* void ∗ *pvData_*)

Mailbox_Receive.

See also

void Mailbox::Receive(void ∗pvData_)

Parameters

handle	Handle of the mailbox object
*pvData_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.

20.125.2.47   void Mailbox_ReceiveTail (Mailbox_t *handle,* void ∗ *pvData_*)

Mailbox_ReceiveTail.

See also

void Mailbox::ReceiveTail(void ∗pvData_)

Parameters

handle	Handle of the mailbox object
*pvData_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.

20.125.2.48   bool Mailbox_Send (Mailbox_t *handle,* void ∗ *pvData_*)

Mailbox_Send.

See also

bool Mailbox::Send(void ∗pvData_)

Parameters

handle	Handle of the mailbox object
*pvData_*	Pointer to the data object to send to the mailbox.

Returns

true - envelope was delivered, false - mailbox is full.

20.125.2.49   bool Mailbox_SendTail (Mailbox_t *handle,* void ∗ *pvData_*)

Mailbox_SendTail.

See also

bool Mailbox::SendTail(void ∗pvData_)

Parameters

handle	Handle of the mailbox object
*pvData_*	Pointer to the data object to send to the mailbox.

Returns

true - envelope was delivered, false - mailbox is full.

20.125.2.50 bool Mailbox_TimedReceive (Mailbox_t *handle,* void ∗ *pvData_,* uint32_t *u32TimeoutMS_*)

Mailbox_TimedReceive.

See also

bool Mailbox::Receive(void ∗pvData_, uint32_t u32TimeoutMS_)

Parameters

handle	Handle of the mailbox object
*pvData_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
*u32TimeoutM↩S_*	Maximum time to wait for delivery.

Returns

true - envelope was delivered, false - delivery timed out.

20.125.2.51 bool Mailbox_TimedReceiveTail (Mailbox_t *handle,* void ∗ *pvData_,* uint32_t *u32TimeoutMS_*)

Mailbox_TimedReceiveTail.

See also

bool Mailbox::ReceiveTail(void ∗pvData_, uint32_t u32TimeoutMS_)

Parameters

handle	Handle of the mailbox object
*pvData_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.
*u32TimeoutM↩S_*	Maximum time to wait for delivery.

Returns

true - envelope was delivered, false - delivery timed out.

20.125.2.52 bool Mailbox_TimedSend (Mailbox_t *handle,* void ∗ *pvData_,* uint32_t *u32TimeoutMS_*)

Mailbox_TimedSend.

See also

bool Mailbox::Send(void ∗pvData_, uint32_t u32TimeoutMS_)

Parameters

handle	Handle of the mailbox object
*pvData_*	Pointer to the data object to send to the mailbox.
*u32TimeoutM↩ S_*	Maximum time to wait for a free transmit slot

Returns

true - envelope was delivered, false - mailbox is full.

20.125.2.53   bool Mailbox_TimedSendTail (Mailbox_t *handle,* **void ∗** *pvData_,* **uint32_t** *u32TimeoutMS_* **)**

Mailbox_TimedSendTail.

See also

bool Mailbox::Send(void ∗pvData_, uint32_t u32TimeoutMS_)

Parameters

handle	Handle of the mailbox object
*pvData_*	Pointer to the data object to send to the mailbox.
*u32TimeoutM↩ S_*	Maximum time to wait for a free transmit slot

Returns

true - envelope was delivered, false - mailbox is full.

20.125.2.54   uint16_t Message_GetCode (Message_t *handle* **)**

Message_GetCode.

See also

uint16_t Message::GetCode()

Parameters

handle	Handle of the message object

Returns

user code set in the object

20.125.2.55   void∗ Message_GetData (Message_t *handle* **)**

Message_GetData.

See also

void∗ Message::GetData()

Parameters

handle	Handle of the message object

Returns

Pointer to the data set in the message object

20.125.2.56 void Message_Init (Message_t *handle*)

Message_Init.

See also

void Message::Init()

Parameters

handle	Handle of the message object

20.125.2.57 void Message_SetCode (Message_t *handle,* uint16_t *u16Code_*)

Message_SetCode.

See also

void Message::SetCode(uint16_t u16Code_)

Parameters

handle	Handle of the message object
*u16Code_*	Data code to set in the object

20.125.2.58 void Message_SetData (Message_t *handle,* void ∗ *pvData_*)

Message_SetData.

See also

void Message::SetData(void ∗pvData_)

Parameters

handle	Handle of the message object
*pvData_*	Pointer to the data object to send in the message

20.125.2.59 uint16_t MessageQueue_GetCount (void)

MessageQueue_GetCount.

See also

uint16_t MessageQueue::GetCount()

Returns

Count of pending messages in the queue.

20.125.2.60 void MessageQueue_Init (MessageQueue_t *handle* **)**

MessageQueue_Init.

See also

> void MessageQueue::Init()

Parameters

handle	Handle to the message queue to initialize

20.125.2.61 Message_t MessageQueue_Receive (MessageQueue_t *handle* **)**

MessageQueue_Receive.

See also

> Message_t MessageQueue::Receive()

Parameters

handle	Handle of the message queue object

Returns

> Pointer to a message object at the head of the queue

20.125.2.62 void MessageQueue_Send (MessageQueue_t *handle,* **Message_t** *hMessage_* **)**

MessageQueue_Send.

See also

> void MessageQueue::Send(Message *pclMessage_)

Parameters

handle	Handle of the message queue object
*hMessage_*	Handle to the message to send to the given queue

20.125.2.63 Message_t MessageQueue_TimedReceive (MessageQueue_t *handle,* **uint32_t** *u32TimeWaitMS_* **)**

MessageQueue_TimedReceive.

See also

> Message_t MessageQueue::TimedReceive(uint32_t u32TimeWaitMS_)

Parameters

handle	Handle of the message queue object
u32TimeWaitM← *S_*	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.

20.125.2.64 void Mutex_Claim (Mutex_t *handle*)

Mutex_Claim.

See also

void Mutex::Claim()

Parameters

handle	Handle of the mutex

20.125.2.65 void Mutex_Init (Mutex_t *handle*)

Mutex_Init.

See also

void Mutex::Init()

Parameters

handle	Handle of the mutex

20.125.2.66 void Mutex_Release (Mutex_t *handle*)

Mutex_Release.

See also

void Mutex::Release()

Parameters

handle	Handle of the mutex

20.125.2.67 bool Mutex_TimedClaim (Mutex_t *handle,* uint32_t *u32WaitTimeMS_*)

Mutex_TimedClaim.

See also

bool Mutex::Claim(uint32_t u32WaitTimeMS_)

Parameters

handle	Handle of the mutex
*u32WaitTimeM↩ S_*	Time to wait before aborting

Returns

true if mutex was claimed, false on timeout

20.125.2.68 void Notify_Init (Notify_t *handle*)

Notify_Init.

See also

void Notify::Init()

Parameters

handle	Handle of the notification object

20.125.2.69 void Notify_Signal (Notify_t *handle*)

Notify_Signal.

See also

void Notify::Signal()

Parameters

handle	Handle of the notification object

20.125.2.70 bool Notify_TimedWait (Notify_t *handle,* uint32_t *u32WaitTimeMS_,* bool ∗ *pbFlag_*)

Notify_TimedWait.

See also

bool Notify::Wait(uint32_t u32WaitTimeMS_, bool ∗pbFlag_)

Parameters

handle	Handle of the notification object
*u32WaitTimeM↩ S_*	Maximum time to wait for notification in ms
*pbFlag_*	Flag to set to true on notification

Returns

true on unblock, false on timeout

20.125.2.71 **void Notify_Wait (Notify_t** *handle,* **bool** ∗ *pbFlag_* **)**

Notify_Wait.

See also

void [Notify::Wait(bool ∗pbFlag_)](#)

Parameters

handle	Handle of the notification object
*pbFlag_*	Flag to set to true on notification

20.125.2.72 **void Scheduler_Enable (bool** *bEnable_* **)**

Scheduler_Enable.

See also

void [Scheduler::SetScheduler(bool bEnable_)](#)

Parameters

bEnable_true	to enable, false to disable the scheduler

20.125.2.73 **Thread_t Scheduler_GetCurrentThread (void)**

Scheduler_GetCurrentThread.

See also

Thread∗ [Scheduler::GetCurrentThread()](#)

Returns

Handle of the currently-running thread

20.125.2.74 **bool Scheduler_IsEnabled (void)**

Scheduler_IsEnabled.

See also

bool [Scheduler::IsEnabled()](#)

Returns

true - scheduler enabled, false - disabled

20.125.2.75 **void Semaphore_Init (Semaphore_t** *handle,* **uint16_t** *u16InitVal_,* **uint16_t** *u16MaxVal_* **)**

Semaphore_Init.

See also

void [Semaphore::Init(uint16_t u16InitVal_, uint16_t u16MaxVal_)](#)

Parameters

handle	Handle of the semaphore
*u16InitVal_*	Initial value of the semaphore
*u16MaxVal_*	Maximum value that can be held for a semaphore

20.125.2.76 void Semaphore_Pend (Semaphore_t *handle*)

Semaphore_Pend.

See also

void Semaphore::Pend()

Parameters

handle	Handle of the semaphore

20.125.2.77 void Semaphore_Post (Semaphore_t *handle*)

Semaphore_Post.

See also

void Semaphore::Post()

Parameters

handle	Handle of the semaphore

20.125.2.78 bool Semaphore_TimedPend (Semaphore_t *handle,* uint32_t *u32WaitTimeMS_*)

Semaphore_TimedPend.

See also

bool Semaphore::Pend(uint32_t u32WaitTimeMS_)

Parameters

handle	Handle of the semaphore
u32WaitTimeM← *S_*	Time in ms to wait

Returns

true if semaphore was acquired, false on timeout

20.125.2.79 void Thread_Exit (Thread_t *handle*)

Thread_Exit.

See also

void Thread::Exit()

Parameters

handle	Handle of the thread

20.125.2.80 PORT_PRIO_TYPE Thread_GetCurPriority (Thread_t *handle*)

Thread_GetCurPriority.

See also

PORT_PRIO_TYPE Thread::GetCurPriority()

Parameters

handle	Handle of the thread

Returns

Current priority of the thread considering priority inheritence

20.125.2.81 uint8_t Thread_GetID (Thread_t *handle*)

Thread_GetID.

See also

uint8_t Thread::GetID()

Parameters

handle	Handle of the thread

Returns

Return ID assigned to the thread

20.125.2.82 PORT_PRIO_TYPE Thread_GetPriority (Thread_t *handle*)

Thread_GetPriority.

See also

PORT_PRIO_TYPE Thread::GetPriority()

Parameters

handle	Handle of the thread

Returns

Current priority of the thread not considering priority inheritence

20.125.2.83 uint16_t Thread_GetQuantum (Thread_t *handle*)

Thread_GetQuantum.

See also

uint16_t Thread::GetQuantum()

Parameters

handle	Handle of the thread

Returns

Thread's current execution quantum

20.125.2.84 uint16_t Thread_GetStackSlack (Thread_t *handle*)

Thread_GetStackSlack.

See also

uint16_t Thread::GetStackSlack()

Parameters

handle	Handle of the thread

Returns

Return the amount of unused stack on the given thread

20.125.2.85 ThreadState_t Thread_GetState (Thread_t *handle*)

Thread_GetState.

See also

ThreadState_t Thread::GetState()

Parameters

handle	Handle of the thread

Returns

The thread's current execution state

20.125.2.86 void Thread_Init (Thread_t *handle,* K_WORD ∗ *pwStack_,* uint16_t *u16StackSize_,* PORT_PRIO_TYPE *uXPriority_,* ThreadEntry_t *pfEntryPoint_,* void ∗ *pvArg_*)

Thread_Init.

See also

void Thread::Init(K_WORD ∗pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_, Thread↩
Entry_t pfEntryPoint_, void ∗pvArg_)

Parameters

handle	Handle of the thread to initialize
*pwStack_*	Pointer to the stack to use for the thread
*u16StackSize_*	Size of the stack (in bytes)
*uXPriority_*	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.

20.125.2.87 void Thread_SetID (Thread_t *handle,* uint8_t *u8ID_*)

Thread_SetID.

See also

void Thread::SetID(uint8_t u8ID_)

Parameters

handle	Handle of the thread
*u8ID_*	ID To assign to the thread

20.125.2.88 void Thread_SetPriority (Thread_t *handle,* PORT_PRIO_TYPE *uXPriority_*)

Thread_SetPriority.

See also

void Thread::SetPriority(PORT_PRIO_TYPE uXPriority_)

Parameters

handle	Handle of the thread
*uXPriority_*	New priority level

20.125.2.89 void Thread_SetQuantum (Thread_t *handle,* uint16_t *u16Quantum_*)

Thread_SetQuantum.

See also

void Thread::SetQuentum(uint16_t u16Quantum_)

Parameters

handle	Handle of the thread
*u16Quantum_*	Time (in ticks) to set for the thread execution quantum

20.125.2.90 void Thread_Sleep (uint32_t *u32TimeMs_*)

Thread_Sleep.

See also

void Thread::Sleep(uint32_t u32TimeMs_)

Parameters

*u32TimeMs_*	Time in ms to block the thread for

20.125.2.91   void Thread_Start (Thread_t *handle*)

Thread_Start.

See also

> void [Thread::Start()](#)

Parameters

handle	Handle of the thread to start

20.125.2.92   void Thread_Stop (Thread_t *handle*)

Thread_Stop.

See also

> void [Thread::Stop()](#)

Parameters

handle	Handle of the thread to stop

20.125.2.93   void Thread_USleep (uint32_t *u32TimeUs_*)

Thread_USleep.

See also

> void [Thread::USleep(uint32_t u32TimeUs_)](#)

Parameters

*u32TimeUs_*	Time in us to block the thread for

20.125.2.94   void Thread_Yield (void)

Thread_Yield.

See also

> void [Thread::Yield()](#)

20.125.2.95   void Timer_Init (Timer_t *handle*)

Timer_Init.

See also

> void Timer::Init()

Parameters

handle	Handle of the timer

20.125.2.96 void Timer_Restart (Timer_t *handle*)

Timer_Restart.

See also

void Timer::Start()

Parameters

handler	Handle of the timer to restart.

20.125.2.97 void Timer_Start (Timer_t *handle,* bool *bRepeat_,* uint32_t *u32IntervalMs_,* uint32_t *u32ToleranceMs_,* TimerCallbackC_t *pfCallback_,* void ∗ *pvData_*)

Timer_Start.

See also

void Timer::Start(bool bRepeat_, uint32_t u32IntervalMs_, uint32_t u32ToleranceMs_, TimerCallbackC_t pf↩
Callback_, void ∗pvData_)

Parameters

handle	Handle of the timer
*bRepeat_*	Restart the timer continuously on expiry
u32IntervalMs↩ *_*	Time in ms to expiry
u32Tolerance↩ *Ms_*	Group with other timers if they expire within the amount of time specified
*pfCallback_*	Callback to run on timer expiry
*pvData_*	Data to pass to the callback on expiry

20.125.2.98 void Timer_Stop (Timer_t *handle*)

Timer_Stop.

See also

void Timer::Stop()

Parameters

handle	Handle of the timer

20.126 mark3c.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
00004 |    \  /   | ||     \    | ||    |    | ||   |/ /      ||___   |
```

```
00005 |     \/    |  ||    \      ||      \      ||      \     ||___    |
00006 |__/\__/|__|_||__|\__\   __||__|\__\   __||__|\__\   __||_____|
00007        |_____|          |_____|          |_____|         |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
00012 See license.txt for more information
00013 =========================================================================*/
00021 #include "mark3cfg.h"
00022 #include "kerneltypes.h"
00023 #include "fake_types.h"
00024 #include "driver3c.h"
00025
00026 #include <stdint.h>
00027 #include <stdbool.h>
00028
00029 #ifndef __MARK3C_H__
00030 #define __MARK3C_H__
00031
00032 #if defined(__cplusplus)
00033 extern "C" {
00034 #endif
00035
00036 //---------------------------------------------------------------------------
00037 // Define a series of handle types to be used in place of the underlying classes
00038 // of Mark3.
00039 typedef void* EventFlag_t;
00040 typedef void* Mailbox_t;
00041 typedef void* Message_t;
00042 typedef void* MessageQueue_t;
00043 typedef void* Mutex_t;
00044 typedef void* Notify_t;
00045 typedef void* Semaphore_t;
00046 typedef void* Thread_t;
00047 typedef void* Timer_t;
00048
00049 //---------------------------------------------------------------------------
00050 // Function pointer types used by Kernel APIs
00051 typedef void (*thread_create_callout_t)(Thread_t hThread_);
00052 typedef void (*thread_exit_callout_t)(Thread_t hThread_);
00053 typedef void (*thread_context_callout_t)(Thread_t hThread_);
00054
00055 //---------------------------------------------------------------------------
00056 // Use the sizes of the structs in fake_types.h to generate opaque object-blobs
00057 // that get instantiated as kernel objects (from the C++ code) later.
00058 #define THREAD_SIZE (sizeof(Fake_Thread))
00059 #define TIMER_SIZE (sizeof(Fake_Timer))
00060 #define SEMAPHORE_SIZE (sizeof(Fake_Semaphore))
00061 #define MUTEX_SIZE (sizeof(Fake_Mutex))
00062 #define MESSAGE_SIZE (sizeof(Fake_Message))
00063 #define MESSAGEQUEUE_SIZE (sizeof(Fake_MessageQueue))
00064 #define MAILBOX_SIZE (sizeof(Fake_Mailbox))
00065 #define NOTIFY_SIZE (sizeof(Fake_Notify))
00066 #define EVENTFLAG_SIZE (sizeof(Fake_EventFlag))
00067
00068 //---------------------------------------------------------------------------
00069 // Macros for declaring opaque buffers of an appropriate size for the given
00070 // kernel objects
00071 #define TOKEN_1(x, y) x##y
00072 #define TOKEN_2(x, y) TOKEN_1(x, y)
00073
00074 // Ensure that opaque buffers are sized to the nearest word - which is
00075 // a platform-dependent value.
00076 #define WORD_ROUND(x) (((x) + (sizeof(K_WORD) - 1)) / sizeof(K_WORD))
00077
00078 #define DECLARE_THREAD(name)                                              \
00079     K_WORD   TOKEN_2(__thread_, name)[WORD_ROUND(THREAD_SIZE)];            \
00080     Thread_t name = (Thread_t)TOKEN_2(__thread_, name);
00081
00082 #define DECLARE_TIMER(name)                                               \
00083     K_WORD  TOKEN_2(__timer_, name)[WORD_ROUND(TIMER_SIZE)];               \
00084     Timer_t name = (Timer_t)TOKEN_2(__timer_, name);
00085
00086 #define DECLARE_SEMAPHORE(name)                                           \
00087     K_WORD      TOKEN_2(__semaphore_, name)[WORD_ROUND(SEMAPHORE_SIZE)];   \
00088     Semaphore_t name = (Semaphore_t)TOKEN_2(__semaphore_, name);
00089
00090 #define DECLARE_MUTEX(name)                                               \
00091     K_WORD  TOKEN_2(__mutex_, name)[WORD_ROUND(MUTEX_SIZE)];
```

```
              \
00092     Mutex_t name = (Mutex_t)TOKEN_2(__mutex_, name);
00093
00094 #define DECLARE_MESSAGE(name)
                      \
00095     K_WORD    TOKEN_2(__message_, name)[WORD_ROUND(MESSAGE_SIZE)];
                  \
00096     Message_t name = (Message_t)TOKEN_2(__message_, name);
00097
00098 #define DECLARE_MESSAGEQUEUE(name)
                          \
00099     K_WORD         TOKEN_2(__messagequeue_, name)[WORD_ROUND(MESSAGEQUEUE_SIZE)];
                      \
00100     MessageQueue_t name = (MessageQueue_t)TOKEN_2(__messagequeue_, name);
00101
00102 #define DECLARE_MAILBOX(name)
                      \
00103     K_WORD    TOKEN_2(__mailbox_, name)[WORD_ROUND(MAILBOX_SIZE)];
                  \
00104     Mailbox_t name = (Mailbox_t)TOKEN_2(__mailbox_, name);
00105
00106 #define DECLARE_NOTIFY(name)
                  \
00107     K_WORD   TOKEN_2(__notify_, name)[WORD_ROUND(NOTIFY_SIZE)];
                  \
00108     Notify_t name = (Notify_t)TOKEN_2(__notify_, name);
00109
00110 #define DECLARE_EVENTFLAG(name)
                      \
00111     K_WORD       TOKEN_2(__eventflag_, name)[WORD_ROUND(EVENTFLAG_SIZE)];
                  \
00112     EventFlag_t name = (EventFlag_t)TOKEN_2(__eventflag_, name);
00113
00114 //-----------------------------------------------------------------------
00115 // Allocate-once Memory managment APIs
00116 #if defined KERNEL_USE_AUTO_ALLOC
00117
00123 void* AutoAlloc(uint16_t u16Size_);
00124 #if KERNEL_USE_SEMAPHORE
00125
00130 Semaphore_t Alloc_Semaphore(void);
00131 #endif
00132 #if KERNEL_USE_MUTEX
00133
00138 Mutex_t Alloc_Mutex(void);
00139 #endif
00140 #if KERNEL_USE_EVENTFLAG
00141
00146 EventFlag_t Alloc_EventFlag(void);
00147 #endif
00148 #if KERNEL_USE_MESSAGE
00149
00154 Message_t Alloc_Message(void);
00160 MessageQueue_t Alloc_MessageQueue(void);
00161 #endif
00162 #if KERNEL_USE_NOTIFY
00163
00168 Notify_t Alloc_Notify(void);
00169 #endif
00170 #if KERNEL_USE_MAILBOX
00171
00176 Mailbox_t Alloc_Mailbox(void);
00177 #endif
00178
00183 Thread_t Alloc_Thread(void);
00184 #if KERNEL_USE_TIMERS
00185
00190 Timer_t Alloc_Timer(void);
00191 #endif
00192 #endif
00193
00194 //-----------------------------------------------------------------------
00195 // Kernel APIs
00200 void Kernel_Init(void);
00205 void Kernel_Start(void);
00212 bool Kernel_IsStarted(void);
00218 void Kernel_SetPanic(PanicFunc_t pfPanic_);
00224 bool Kernel_IsPanic(void);
00230 void Kernel_Panic(uint16_t u16Cause_);
00231 #if KERNEL_USE_IDLE_FUNC
00232
00237 void Kernel_SetIdleFunc(IdleFunc_t pfIdle_);
00238 #endif
00239
00240 #if KERNEL_USE_THREAD_CALLOUTS
00241
00246 void Kernel_SetThreadCreateCallout(thread_create_callout_t pfCreate_);
```

```
00252 void Kernel_SetThreadExitCallout(thread_exit_callout_t pfExit_);
00253
00259 void Kernel_SetThreadContextSwitchCallout(thread_context_callout_t
      pfContext_);
00260
00266 thread_create_callout_t Kernel_GetThreadCreateCallout(void);
00267
00273 thread_exit_callout_t Kernel_GetThreadExitCallout(void);
00274
00280 thread_context_callout_t Kernel_GetThreadContextSwitchCallout(void);
00281 #endif
00282
00283 #if KERNEL_USE_STACK_GUARD
00284
00290 static void Kernel_SetStackGuardThreshold(uint16_t u16Threshold_);
00291
00297 static uint16_t Kernel_GetStackGuardThreshold(void);
00298 #endif
00299 //---------------------------------------------------------------------------
00300 // Scheduler APIs
00306 void Scheduler_Enable(bool bEnable_);
00312 bool Scheduler_IsEnabled(void);
00318 Thread_t Scheduler_GetCurrentThread(void);
00319
00320 //---------------------------------------------------------------------------
00321 // Thread APIs
00335 void Thread_Init(Thread_t        handle,
00336                  K_WORD*         pwStack_,
00337                  uint16_t        u16StackSize_,
00338                  PORT_PRIO_TYPE  uXPriority_,
00339                  ThreadEntry_t pfEntryPoint_,
00340                  void*           pvArg_);
00346 void Thread_Start(Thread_t handle);
00352 void Thread_Stop(Thread_t handle);
00353 #if KERNEL_USE_THREADNAME
00354
00360 void Thread_SetName(Thread_t handle, const char* szName_);
00367 const char* Thread_GetName(Thread_t handle);
00368 #endif
00369
00375 PORT_PRIO_TYPE Thread_GetPriority(Thread_t handle);
00382 PORT_PRIO_TYPE Thread_GetCurPriority(Thread_t handle);
00383 #if KERNEL_USE_QUANTUM
00384
00390 void Thread_SetQuantum(Thread_t handle, uint16_t u16Quantum_);
00397 uint16_t Thread_GetQuantum(Thread_t handle);
00398 #endif
00399
00405 void Thread_SetPriority(Thread_t handle, PORT_PRIO_TYPE uXPriority_);
00406 #if KERNEL_USE_DYNAMIC_THREADS
00407
00412 void Thread_Exit(Thread_t handle);
00413 #endif
00414 #if KERNEL_USE_SLEEP
00415
00420 void Thread_Sleep(uint32_t u32TimeMs_);
00426 void Thread_USleep(uint32_t u32TimeUs_);
00427 #endif
00428
00432 void Thread_Yield(void);
00439 void Thread_SetID(Thread_t handle, uint8_t u8ID_);
00446 uint8_t Thread_GetID(Thread_t handle);
00453 uint16_t Thread_GetStackSlack(Thread_t handle);
00460 ThreadState_t Thread_GetState(Thread_t handle);
00461
00462 //---------------------------------------------------------------------------
00463 // Timer APIs
00464 #if KERNEL_USE_TIMERS
00465 typedef void (*TimerCallbackC_t)(Thread_t hOwner_, void* pvData_);
00471 void Timer_Init(Timer_t handle);
00483 void Timer_Start(Timer_t         handle,
00484                  bool            bRepeat_,
00485                  uint32_t        u32IntervalMs_,
00486                  uint32_t        u32ToleranceMs_,
00487                  TimerCallbackC_t pfCallback_,
00488                  void*           pvData_);
00489
00495 void Timer_Restart(Timer_t handle);
00496
00502 void Timer_Stop(Timer_t handle);
00503 #endif
00504
00505 //---------------------------------------------------------------------------
00506 // Semaphore APIs
00507 #if KERNEL_USE_SEMAPHORE
00508
00515 void Semaphore_Init(Semaphore_t handle, uint16_t u16InitVal_, uint16_t u16MaxVal_);
```

```
00521 void Semaphore_Post(Semaphore_t handle);
00527 void Semaphore_Pend(Semaphore_t handle);
00528 #if KERNEL_USE_TIMEOUTS
00529
00536 bool Semaphore_TimedPend(Semaphore_t handle, uint32_t u32WaitTimeMS_);
00537 #endif
00538 #endif
00539
00540 //-----------------------------------------------------------------------------
00541 // Mutex APIs
00542 #if KERNEL_USE_MUTEX
00543
00548 void Mutex_Init(Mutex_t handle);
00554 void Mutex_Claim(Mutex_t handle);
00560 void Mutex_Release(Mutex_t handle);
00561 #if KERNEL_USE_TIMEOUTS
00562
00569 bool Mutex_TimedClaim(Mutex_t handle, uint32_t u32WaitTimeMS_);
00570 #endif
00571 #endif
00572
00573 //-----------------------------------------------------------------------------
00574 // EventFlag APIs
00575 #if KERNEL_USE_EVENTFLAG
00576
00581 void EventFlag_Init(EventFlag_t handle);
00590 uint16_t EventFlag_Wait(EventFlag_t handle, uint16_t u16Mask_,
      EventFlagOperation_t eMode_);
00591 #if KERNEL_USE_TIMEOUTS
00592
00601 uint16_t EventFlag_TimedWait(EventFlag_t handle, uint16_t u16Mask_,
      EventFlagOperation_t eMode_, uint32_t u32TimeMS_);
00602 #endif
00603
00609 void EventFlag_Set(EventFlag_t handle, uint16_t u16Mask_);
00616 void EventFlag_Clear(EventFlag_t handle, uint16_t u16Mask_);
00623 uint16_t EventFlag_GetMask(EventFlag_t handle);
00624 #endif
00625
00626 //-----------------------------------------------------------------------------
00627 // Notification APIs
00628 #if KERNEL_USE_NOTIFY
00629
00634 void Notify_Init(Notify_t handle);
00640 void Notify_Signal(Notify_t handle);
00647 void Notify_Wait(Notify_t handle, bool* pbFlag_);
00648 #if KERNEL_USE_TIMEOUTS
00649
00657 bool Notify_TimedWait(Notify_t handle, uint32_t u32WaitTimeMS_, bool* pbFlag_);
00658 #endif
00659 #endif
00660
00661 //-----------------------------------------------------------------------------
00662 // Atomic Functions
00663 #if KERNEL_USE_ATOMIC
00664
00671 uint8_t Atomic_Set8(uint8_t* pu8Source_, uint8_t u8Val_);
00679 uint16_t Atomic_Set16(uint16_t* pu16Source_, uint16_t u16Val_);
00687 uint32_t Atomic_Set32(uint32_t* pu32Source_, uint32_t u32Val_);
00695 uint8_t Atomic_Add8(uint8_t* pu8Source_, uint8_t u8Val_);
00703 uint16_t Atomic_Add16(uint16_t* pu16Source_, uint16_t u16Val_);
00711 uint32_t Atomic_Add32(uint32_t* pu32Source_, uint32_t u32Val_);
00719 uint8_t Atomic_Sub8(uint8_t* pu8Source_, uint8_t u8Val_);
00727 uint16_t Atomic_Sub16(uint16_t* pu16Source_, uint16_t u16Val_);
00735 uint32_t Atomic_Sub32(uint32_t* pu32Source_, uint32_t u32Val_);
00744 bool Atomic_TestAndSet(bool* pbLock);
00745 #endif
00746
00747 //-----------------------------------------------------------------------------
00748 // Message/Message Queue APIs
00749 #if KERNEL_USE_MESSAGE
00750
00755 void Message_Init(Message_t handle);
00762 void Message_SetData(Message_t handle, void* pvData_);
00769 void* Message_GetData(Message_t handle);
00776 void Message_SetCode(Message_t handle, uint16_t u16Code_);
00783 uint16_t Message_GetCode(Message_t handle);
00789 void GlobalMessagePool_Push(Message_t handle);
00795 Message_t GlobalMessagePool_Pop(void);
00801 void MessageQueue_Init(MessageQueue_t handle);
00808 Message_t MessageQueue_Receive(MessageQueue_t handle);
00809 #if KERNEL_USE_TIMEOUTS
00810
00820 Message_t MessageQueue_TimedReceive(MessageQueue_t handle, uint32_t u32TimeWaitMS_
      );
00821 #endif
00822
```

```
00829 void MessageQueue_Send(MessageQueue_t handle, Message_t hMessage_);
00830
00836 uint16_t MessageQueue_GetCount(void);
00837 #endif
00838
00839 //---------------------------------------------------------------------------
00840 // Mailbox APIs
00841 #if KERNEL_USE_MAILBOX
00842
00851 void Mailbox_Init(Mailbox_t handle, void* pvBuffer_, uint16_t u16BufferSize_, uint16_t
      u16ElementSize_);
00852
00860 bool Mailbox_Send(Mailbox_t handle, void* pvData_);
00861
00869 bool Mailbox_SendTail(Mailbox_t handle, void* pvData_);
00870
00879 bool Mailbox_TimedSend(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);
00880
00889 bool Mailbox_TimedSendTail(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);
00890
00898 void Mailbox_Receive(Mailbox_t handle, void* pvData_);
00899
00907 void Mailbox_ReceiveTail(Mailbox_t handle, void* pvData_);
00908 #if KERNEL_USE_TIMEOUTS
00909
00919 bool Mailbox_TimedReceive(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);
00920
00930 bool Mailbox_TimedReceiveTail(Mailbox_t handle, void* pvData_, uint32_t
      u32TimeoutMS_);
00931
00938 uint16_t Mailbox_GetFreeSlots(Mailbox_t handle);
00939
00946 bool Mailbox_IsFull(Mailbox_t handle);
00947
00954 bool Mailbox_IsEmpty(Mailbox_t handle);
00955 #endif
00956 #endif
00957
00958 //---------------------------------------------------------------------------
00959 // Kernel-Aware Simulation APIs
00960 #if KERNEL_AWARE_SIMULATION
00961
00967 void KernelAware_ProfileInit(const char* szStr_);
00968
00973 void KernelAware_ProfileStart(void);
00974
00979 void KernelAware_ProfileStop(void);
00980
00985 void KernelAware_ProfileReport(void);
00986
00992 void KernelAware_ExitSimulator(void);
00993
00999 void KernelAware_Print(const char* szStr_);
01000
01007 void KernelAware_Trace(uint16_t u16File_, uint16_t u16Line_);
01008
01016 void KernelAware_Trace1(uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_);
01025 void KernelAware_Trace2(uint16_t u16File_, uint16_t u16Line_, uint16_t u16Arg1_, uint16_t
      u16Arg2_);
01035 bool KernelAware_IsSimulatorAware(void);
01036 #endif
01037
01038 #if defined(__cplusplus)
01039 }
01040 #endif
01041
01042 #endif // __MARK3C_H__
```

Chapter 21

Example Documentation

21.1 buffalogger/main.cpp

This example demonstrates how low-overhead logging can be implemented using buffalogger.

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
|    \  /  | ||    \    ||    |    ||    ||  /  /    ||___  |
|     \/   | ||     \    ||    \    ||    ||_/  /     ||___  |
|__/\__/|__|_||__|\__\  _||__|\__\  _||__|\__\  _||_____|
    |_____|        |_____|        |_____|        |_____|
--[Mark3 Realtime Platform]-------------------------------------------------

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
#include "kerneldebug.h"
#include "drvUART.h"
#include "tracebuffer.h"
#include "ksemaphore.h"

/*===========================================================================

Example - Logging data via buffalogger/debug APIs.

===========================================================================*/

#if !(KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION)
# error "Buffalogger demo requires tracebuffer support"
#endif

#if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION


#define _CAN_HAS_DEBUG
//--[Autogenerated - Do Not Modify]------------------------------------------
#include "dbg_file_list.h"
#include "buffalogger.h"
#if defined(DBG_FILE)
#error "Debug logging file token already defined!  Bailing."
#else
#define DBG_FILE _DBG___EXAMPLES_AVR_BUFFALOGGER_MAIN_CPP
#endif
//--[End Autogenerated content]----------------------------------------------

//---------------------------------------------------------------------------
// This block declares the thread data for the main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP_STACK_SIZE (192 / sizeof(K_WORD))
static Thread clAppThread;
static K_WORD awAppStack[APP_STACK_SIZE];
static void AppMain(void* unused_);

#define IDLE_STACK_SIZE (192 / sizeof(K_WORD))
static Thread clIdleThread;
static K_WORD awIdleStack[APP_STACK_SIZE];
static void IdleMain(void* unused_);
```

```
#define LOGGER_STACK_SIZE (192 / sizeof(K_WORD))
static Thread clLoggerThread;
static K_WORD awLoggerStack[APP_STACK_SIZE];
static void LoggerMain(void* unused_);
static volatile bool bPingPong;
static Semaphore    clSem;

//---------------------------------------------------------------------------
static ATMegaUART clUART;

//---------------------------------------------------------------------------
#define UART_SIZE_TX (32)
#define UART_SIZE_RX (8)

static uint8_t aucTxBuffer[UART_SIZE_TX];
static uint8_t aucRxBuffer[UART_SIZE_RX];

static volatile uint16_t* pu16Log;
static volatile uint16_t  u16LogLen;

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

void IdleMain(void* unused_)
{
    while (1) {
    }
}
//---------------------------------------------------------------------------
void LoggerCallback(uint16_t* pu16Data_, uint16_t u16Len_, bool bPingPong_)
{
    CS_ENTER();
    bPingPong = bPingPong_;
    pu16Log   = pu16Data_;
    u16LogLen = u16Len_;
    CS_EXIT();

    clSem.Post();
}

//---------------------------------------------------------------------------
void LoggerMain(void* unused_)
{
    while (1) {
        uint8_t* src;
        uint16_t len;

        clSem.Pend();

        CS_ENTER();
        src = (uint8_t*)pu16Log;
        len = u16LogLen * sizeof(uint16_t);
        CS_EXIT();

        uint16_t written = 0;
        while (len != written) {
            written += clUART.Write(len - written, src + written);
        }
    }
}

//---------------------------------------------------------------------------
int main(void)
{
    Kernel::Init();

    // Example assumes use of built-in idle.
    clAppThread.Init(awAppStack, APP_STACK_SIZE, 2, AppMain, 0);
    clAppThread.Start();

    clLoggerThread.Init(awLoggerStack, LOGGER_STACK_SIZE, 1, LoggerMain, 0);
    clLoggerThread.Start();

    clIdleThread.Init(awIdleStack, IDLE_STACK_SIZE, 0, IdleMain, 0);
    clIdleThread.Start();

    clUART.SetName("/dev/tty");
    clUART.Init();
    clUART.Open();

    DriverList::Add(&clUART);
```

```
    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
void AppMain(void* unused_)
{
    {
        uint32_t u32Baud = 57600 * 4;
        clUART.Control(CMD_SET_BAUDRATE, &u32Baud, 0, 0, 0);
    }
    clUART.Control(CMD_SET_BUFFERS, (void*)aucRxBuffer, UART_SIZE_RX, (void*)aucTxBuffer, UART_SIZE_TX);

    clSem.Init(0, 1);

    TraceBuffer::SetCallback(LoggerCallback);
    volatile uint16_t u16Iteration = 0;
    while (1) {
        Thread::Sleep(100);
        USER_TRACE("Beginning of the main application loop!");

        Thread::Sleep(100);
        USER_TRACE_1(" Iteration: %d", u16Iteration++);

        Thread::Sleep(100);
        USER_TRACE("End of the main application loop!");
    }
}


#endif //#if KERNEL_USE_DEBUG && !KERNEL_AWARE_SIMULATION
```

21.2 lab10_notifications/main.cpp

This examples demonstrates how to use notifcation objects as a thread synchronization mechanism.

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

     _____        _____        _____        _____
  __|   _|__   __|   _|__   __|   _|__   __|   _|__     ___
 |   \/   |  |     |    |  |     |     |  |   |/ /   |    |
 |        |  |     |     \ |     |     \ |     |     |    |
 |___/\___|__|__|__|__|\__\  __|__|__|\__\  __|__|__|\__\  __|__|_____|
    |_____|        |_____|        |_____|        |_____|

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

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 10:  Thread Notifications

Lessons covered in this example include:
- Create a notification object, and use it to synchronize execution of Threads.

Takeaway:
- Notification objects are a lightweight mechanism to signal thread execution
  in situations where even a semaphore would be a heavier-weigth option.

===========================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
#define APP_STACK_SIZE (256 / sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP_STACK_SIZE];
static void App1Main(void* unused_);

//---------------------------------------------------------------------------
static Thread clApp2Thread;
```

```
static K_WORD awApp2Stack[APP_STACK_SIZE];
static void App2Main(void* unused_);

//---------------------------------------------------------------------------
// Notification object used in the example.
static Notify clNotify;

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    // Initialize notifer and notify-ee threads
    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp1Thread.Start();

    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);
    clApp2Thread.Start();

    // Initialize the Notify objects
    clNotify.Init();

    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    while (1) {
        bool bNotified = false;
        // Block the thread until the notification object is signalled from
        // elsewhere.
        clNotify.Wait(&bNotified);

        KernelAware::Print("T1: Notified\n");
    }
}

//---------------------------------------------------------------------------
void App2Main(void* unused_)
{
    while (1) {
        // Wait a while, then signal the notification object

        KernelAware::Print("T2: Wait 1s\n");
        Thread::Sleep(1000);

        KernelAware::Print("T2: Notify\n");
        clNotify.Signal();
    }
}
```

21.3 lab11_mailboxes/main.cpp

This examples shows how to use mailboxes to deliver data between threads in a synchronized way.

```
/*===========================================================================
     _____        _____        _____        _____
 ___|__   |__  __|__   |__  __|__   |__  __|__   |__  _____
|    \ /    |  |   \      ||      |      ||  |/ /     ||___   |
|     \/    |  |    \     ||      \      ||  |/  \    ||___   |
|__/\__/|__|__||__|\__\  __||__|\__\  __||__|\__\  __||_____|
     |_____|        |_____|        |_____|        |_____|
--[Mark3 Realtime Platform]-------------------------------------------------

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 11:  Mailboxes

Lessons covered in this example include:
- Initialize a mailbox for use as an IPC mechanism.
- Create and use mailboxes to pass data between threads.
```

```
Takeaway:
- Mailboxes are a powerful IPC mechanism used to pass messages of a fixed-size
  between threads.

=======================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
#define APP_STACK_SIZE (256 / sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP_STACK_SIZE];
static void App1Main(void* unused_);

//---------------------------------------------------------------------------
static Thread clApp2Thread;
static K_WORD awApp2Stack[APP_STACK_SIZE];
static void App2Main(void* unused_);

//---------------------------------------------------------------------------
static Mailbox clMailbox;
static uint8_t au8MBData[100];

typedef struct {
    uint8_t au8Buffer[10];
} MBType_t;

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    // Initialize the threads used in this example
    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp1Thread.Start();

    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 2, App2Main, 0);
    clApp2Thread.Start();

    // Initialize the mailbox used in this example
    clMailbox.Init(au8MBData, 100, sizeof(MBType_t));

    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    while (1) {
        MBType_t stMsg;

        // Wait until there is an envelope available in the shared mailbox, and
        // then log a trace message.
        clMailbox.Receive(&stMsg);
        KernelAware::Trace(0, __LINE__, stMsg.au8Buffer[0], stMsg.au8Buffer[9]);
    }
}

//---------------------------------------------------------------------------
void App2Main(void* unused_)
{
    while (1) {
        MBType_t stMsg;

        // Place a bunch of envelopes in the mailbox, and then wait for a
        // while.  Note that this thread has a higher priority than the other
        // thread, so it will keep pushing envelopes to the other thread until
        // it gets to the sleep, at which point the other thread will be allowed
        // to execute.

        KernelAware::Print("Messages Begin\n");

        for (uint8_t i = 0; i < 10; i++) {
            for (uint8_t j = 0; j < 10; j++) {
                stMsg.au8Buffer[j] = (i * 10) + j;
            }
            clMailbox.Send(&stMsg);
```

```
        }

        KernelAware::Print("Messages End\n");
        Thread::Sleep(2000);
    }
}
```

21.4   lab1_kernel_setup/main.cpp

This example demonstrates basic kernel setup with two threads.

```
/*===========================================================================
     _____        _____        _____        _____
  ___|    _|__  __|_    |__    |__    __|__  |__  _____
 |    \  /    |  ||    \       ||       |     ||  |/ /     ||___   |
 |     \/     |  ||     \      ||       \     ||   \       ||___   |
 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
      |_____|        |_____|        |_____|        |_____|

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

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 1: Initializing the Mark3 RTOS kernel with two threads.

The following example code presents a working example of how to initialize
the Mark3 RTOS kernel, configure two application threads, and execute the
configured tasks.  This example also uses the flAVR kernel-aware module to
print out messages when run through the flAVR AVR Simulator.  This is a
turnkey-ready example of how to use the Mark3 RTOS at its simplest level,
and should be well understood before moving on to other examples.

Lessons covered in this example include:

- usage of the Kernel class - configuring and starting the kernel
- usage of the Thread class - initializing and starting static threads.
- Demonstrate the relationship between Thread objects, stacks, and entry
  functions.
- usage of Thread::Sleep() to block execution of a thread for a period of time
- When using an idle thread, the idle thread MUST not block.

Exercise:

- Add another application thread that prints a message, flashes an LED, etc.
  using the code below as an example.

Takeaway:

At the end of this example, the reader should be able to use the Mark3
Kernel and Thread APIs to initialize and start the kernel with any number
of static threads.

===========================================================================*/
extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
// This block declares the thread data for the main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clAppThread;
static K_WORD awAppStack[APP_STACK_SIZE];
static void AppMain(void* unused_);

//---------------------------------------------------------------------------
// This block declares the thread data for the idle thread.  It defines a
// thread object, stack (in word-array form), and the entry-point function
// used by the idle thread.
#define IDLE_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clIdleThread;
static K_WORD awIdleStack[IDLE_STACK_SIZE];
static void IdleMain(void* unused_);
```

```cpp
//---------------------------------------------------------------------------
int main(void)
{
    // Before any Mark3 RTOS APIs can be called, the user must call Kernel::Init().
    // Note that if you have any hardware-specific init code, it can be called
    // before Kernel::Init, so long as it does not enable interrupts, or
    // rely on hardware peripherals (timer, software interrupt, etc.) used by the
    // kernel.
    Kernel::Init();

    // Once the kernel initialization has been complete, the user can add their
    // application thread(s) and idle thread.  Threads added before the kerel
    // is started are refered to as the "static threads" in the system, as they
    // are the default working-set of threads that make up the application on
    // kernel startup.

    // Initialize the application thread to use a specified word-array as its stack.
    // The thread will run at priority level "1", and start execution the
    // "AppMain" function when it's started.
    clAppThread.Init(awAppStack, sizeof(awAppStack), 1, AppMain, 0);

    // Initialize the idle thread to use a specific word-array as its stack.
    // The thread will run at priority level "0", which is reserved for the idle
    // priority thread.  IdleMain will be run when the thread is started.
    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);

    // Once the static threads have been added, the user must then ensure that the
    // threads are ready to execute.  By default, creating a thread is created
    // in a STOPPED state.  All threads must manually be started using the
    // Start() API before they will be scheduled by the system.  Here, we are
    // starting the application and idle threads before starting the kernel - and
    // that's OK.  When the kernel is started, it will choose which thread to run
    // first from the pool of ready threads.

    clAppThread.Start();
    clIdleThread.Start();

    // All threads have been initialized and made ready.  The kernel will now
    // select the first thread to run, enable the hardware required to run the
    // kernel (Timers, software interrupts, etc.), and then do whatever is
    // necessary to maneuver control of thread execution to the kernel.  At this
    // point, execution will transition to the highest-priority ready thread.
    // This function will not return.

    Kernel::Start();

    // As Kernel::Start() results in the operating system being executed, control
    // will not be relinquished back to main().  The "return 0" is simply to
    // avoid warnings.

    return 0;
}

//---------------------------------------------------------------------------
void AppMain(void* unused_)
{
    // This function is run from within the application thread.  Here, we
    // simply print a friendly greeting and allow the thread to sleep for a
    // while before repeating the message.  Note that while the thread is
    // sleeping, CPU execution will transition to the Idle thread.

    while (1) {
        KernelAware::Print("Hello World!\n");
        Thread::Sleep(1000);
    }
}

//---------------------------------------------------------------------------
void IdleMain(void* unused_)
{
    while (1) {
        // Low priority task + power management routines go here.
        // The actions taken in this context must *not* cause the thread
        // to block, as the kernel requires that at least one thread is
        // schedulable at all times when not using an idle thread.

        // Note that if you have no special power-management code or idle
        // tasks, an empty while(1){} loop is sufficient to guarantee that
        // condition.
    }
}
```

21.5 lab2_idle_function/main.cpp

This example demonstrates how to use the idle function, instead of an idle thread to manage system inactivity.

```
/*===========================================================================
      _____        _____        _____        _____
   ___|   _|__   __|_  |__   __|_  |__    __|_  |__   __|_  |_____
  |     \ /   | | |   \       ||         ||   |/ /     ||___    |
  |      \/   | | |    \      ||    \    ||    \       ||___    |
  |__/\__/|__|_||__|\__\   _||__|\__\   _||__|\__\   _||_____|
      |____|        |____|        |____|        |____|

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

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 2: Initializing the Mark3 RTOS kernel with one thread.

The following example code presents a working example of how to initialize
the Mark3 RTOS kernel, configured to use an application thread and the special
Kernel-Idle function.  This example is functionally identical to lab1, although
it uses less memory as a result of only requiring one thread.  This example also
uses the flAVR kernel-aware module to print out messages when run through the
flAVR AVR Simulator.

Lessons covered in this example include:

- usage of the Kernel::SetIdleFunc() API
- Changing an idle thread into an idle function
- You can save a thread and a stack by using an idle function instead of a
  dedicated idle thread.

Takeaway:

The Kernel-Idle context allows you to run the Mark3 RTOS without running
a dedicated idle thread (where supported).  This results in a lower overall
memory footprint for the application, as you can avoid having to declare
a thread object and stack for Idle functionality.

===========================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
// This block declares the thread data for the main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clAppThread;
static K_WORD awAppStack[APP_STACK_SIZE];
static void AppMain(void* unused_);

//---------------------------------------------------------------------------
// This block declares the special function called from with the special
// Kernel-Idle context.  We use the Kernel::SetIdleFunc() API to ensure that
// this function is called to provide our idle context.
static void IdleMain(void);

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in lab1.
    Kernel::Init();

    // Initialize the main application thread, as in lab1.  Note that even
    // though we're using an Idle function and not a dedicated thread, priority
    // level 0 is still reserved for idle functionality.  Application threads
    // should never be scheduled at priority level 0 when the idle function is
    // used instead of an idle thread.
    clAppThread.Init(awAppStack, sizeof(awAppStack), 1, AppMain, 0);
    clAppThread.Start();

    // This function is used to install our specified idle function to be called
```

```
        // whenever there are no ready threads in the system.  Note that if no
        // Idle function is specified, a default will be used.  Note that this default
        // function is essentially a null operation.
        Kernel::SetIdleFunc(IdleMain);

        Kernel::Start();

        return 0;
}

//---------------------------------------------------------------------------
void AppMain(void* unused_)
{
        // Same as in lab1.
        while (1) {
            KernelAware::Print("Hello World!\n");
            Thread::Sleep(1000);
        }
}

//---------------------------------------------------------------------------
void IdleMain(void)
{
        // Low priority task + power management routines go here.
        // The actions taken in this context must *not* cause a blocking call,
        // similar to the requirements for an idle thread.

        // Note that unlike an idle thread, the idle function must run to
        // completion.  As this is also called from a nested interrupt context,
        // it's worthwhile keeping this function brief, limited to absolutely
        // necessary functionality, and with minimal stack use.
}
```

21.6 lab3_round_robin/main.cpp

This example demonstrates how to use round-robin thread scheduling with multiple threads of the same priority.

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| |   |__  _____
|    \  /  |  | |  |  |   |   ||  |  ||  |/ /     ||  |  |
|     \/   |  | |  |  \      ||  |   \     ||  \      ||__   |
|__/\__/|__|__|__|\__\  __||__|\__\  __||__|\__\  __||_____|
     |____|      |____|      |____|      |____|

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

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 3:  using round-robin scheduling to time-slice the CPU.

Lessons covered in this example include:
- Threads at the same priority get timesliced automatically
- The Thread::SetQuantum() API can be used to set the maximum amount of CPU
  time a thread can take before being swapped for another task at that
  priority level.

Takeaway:

- CPU Scheduling can be achieved using not just strict Thread priority, but
  also with round-robin time-slicing between threads at the same priority.

===========================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
```

```cpp
#define APP1_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP1_STACK_SIZE];
static void App1Main(void* unused_);

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp2Thread;
static K_WORD awApp2Stack[APP2_STACK_SIZE];
static void App2Main(void* unused_);

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in lab1.
    Kernel::Init();

    // In this exercise, we create two threads at the same priority level.
    // As a result, the CPU will automatically swap between these threads
    // at runtime to ensure that each get a chance to execute.

    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);

    // Set the threads up so that Thread 1 can get 4ms of CPU time uninterrupted,
    // but Thread 2 can get 8ms of CPU time uninterrupted.  This means that
    // in an ideal situation, Thread 2 will get to do twice as much work as
    // Thread 1 - even though they share the same scheduling priority.

    // Note that if SetQuantum() isn't called on a thread, a default value
    // is set such that each thread gets equal timeslicing in the same
    // priority group by default.  You can play around with these values and
    // observe how it affects the execution of both threads.

    clApp1Thread.SetQuantum(4);
    clApp2Thread.SetQuantum(8);

    clApp1Thread.Start();
    clApp2Thread.Start();

    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    // Simple loop that increments a volatile counter to 1000000 then resets
    // it while printing a message.
    volatile uint32_t u32Counter = 0;
    while (1) {
        u32Counter++;
        if (u32Counter == 1000000) {
            u32Counter = 0;
            KernelAware::Print("Thread 1 - Did some work\n");
        }
    }
}

//---------------------------------------------------------------------------
void App2Main(void* unused_)
{
    // Same as App1Main.  However, as this thread gets twice as much CPU time
    // as Thread 1, you should see its message printed twice as often as the
    // above function.
    volatile uint32_t u32Counter = 0;
    while (1) {
        u32Counter++;
        if (u32Counter == 1000000) {
            u32Counter = 0;
            KernelAware::Print("Thread 2 - Did some work\n");
        }
    }
}
```

21.7 lab4_semaphores/main.cpp

This example demonstrates how to use semaphores for Thread synchronization.

```
/*===========================================================================
     _____        _____        _____        _____
  ___|    _|__  __|__   |__  __|__   |__  __|__   |__  _____
 |    \  /  |  ||    \     ||      |     ||  |/ /     ||___   |
 |     \/   |  ||     \    ||      |     ||  |  \     ||___   |
 |__/\__/|__||__|\__\  _||__|\__\  _||__|\__\  _||_____|
     |____|        |____|        |____|        |____|

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

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 4:  using binary semaphores

In this example, we implement two threads, synchronized using a semaphore to
model the classic producer-consumer pattern.  One thread does work, and then
posts the semaphore indicating that the other thread can consume that work.
The blocking thread just waits idly until there is data for it to consume.

Lessons covered in this example include:
-Use of a binary semaphore to implement the producer-consumer pattern
-Synchronization of threads (within a single priority, or otherwise)
 using a semaphore

Takeaway:

Semaphores can be used to control which threads execute at which time.  This
allows threads to work cooperatively to achieve a goal in the system.

===========================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP1_STACK_SIZE];
static void App1Main(void* unused_);

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp2Thread;
static K_WORD awApp2Stack[APP2_STACK_SIZE];
static void App2Main(void* unused_);

//---------------------------------------------------------------------------
// This is the semaphore that we'll use to synchronize two threads in this
// demo application
static Semaphore clMySem;

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    // In this example we create two threads to illustrate the use of a
    // binary semaphore as a synchronization method between two threads.

    // Thread 1 is a "consumer" thread -- It waits, blocked on the semaphore
    // until thread 2 is done doing some work.  Once the semaphore is posted,
    // the thread is unblocked, and does some work.

    // Thread 2 is thus the "producer" thread -- It does work, and once that
    // work is done, the semaphore is posted to indicate that the other thread
    // can use the producer's work product.

    clApp1Thread.Init(awApp1Stack, APP1_STACK_SIZE, 1, App1Main, 0);
    clApp2Thread.Init(awApp2Stack, APP2_STACK_SIZE, 1, App2Main, 0);
```

```
    clApp1Thread.Start();
    clApp2Thread.Start();

    // Initialize a binary semaphore (maximum value of one, initial value of
    // zero).
    clMySem.Init(0, 1);

    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    while (1) {
        // Wait until the semaphore is posted from the other thread
        KernelAware::Print("Wait\n");
        clMySem.Pend();

        // Producer thread has finished doing its work -- do something to
        // consume its output.  Once again - a contrived example, but we
        // can imagine that printing out the message is "consuming" the output
        // from the other thread.
        KernelAware::Print("Triggered!\n");
    }
}

//---------------------------------------------------------------------------
void App2Main(void* unused_)
{
    volatile uint32_t u32Counter = 0;

    while (1) {
        // Do some work.  Once the work is complete, post the semaphore.  This
        // will cause the other thread to wake up and then take some action.
        // It's a bit contrived, but imagine that the results of this process
        // are necessary to drive the work done by that other thread.
        u32Counter++;
        if (u32Counter == 1000000) {
            u32Counter = 0;
            KernelAware::Print("Posted\n");
            clMySem.Post();
        }
    }
}
```

21.8 lab5_mutexes/main.cpp

This example demonstrates how to use mutexes to protect against concurrent access to resources.

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __|__  __|__   |
|    \  /  |   ||    \      ||     |      ||  |/ /     ||__    |
|     \/   |   ||     \     ||     |      ||     \     ||__    |
|__/\__/|__||__||__|\__\  __||__|\__\  __||__|__|\__\  __||____|
    |_____|        |_____|        |_____|        |_____|
--[Mark3 Realtime Platform]-------------------------------------------------

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 5:  using Mutexes.

Lessons covered in this example include:
-You can use mutexes to lock accesses to a shared resource

Takeaway:

===========================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif

extern "C" {
```

```cpp
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP1_STACK_SIZE];
static void App1Main(void* unused_);

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp2Thread;
static K_WORD awApp2Stack[APP2_STACK_SIZE];
static void App2Main(void* unused_);

//---------------------------------------------------------------------------
// This is the mutex that we'll use to synchronize two threads in this
// demo application.
static Mutex clMyMutex;

// This counter variable is the "shared resource" in the example, protected
// by the mutex.  Only one thread should be given access to the counter at
// any time.
static volatile uint32_t u32Counter = 0;

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);

    clApp1Thread.Start();
    clApp2Thread.Start();

    // Initialize the mutex used in this example.
    clMyMutex.Init();

    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    while (1) {
        // Claim the mutex.  This will prevent any other thread from claiming
        // this lock simulatenously.  As a result, the other thread has to
        // wait until we're done before it can do its work.  You will notice
        // that the Start/Done prints for the thread will come as a pair (i.e.
        // you won't see "Thread2: Start" then "Thread1: Start").

        clMyMutex.Claim();

        // Start our work (incrementing a counter).  Notice that the Start and
        // Done prints wind up as a pair when simuated with flAVR.

        KernelAware::Print("Thread1: Start\n");
        u32Counter++;
        while (u32Counter <= 1000000) {
            u32Counter++;
        }
        u32Counter = 0;
        KernelAware::Print("Thread1: Done\n");

        // Release the lock, allowing the other thread to do its thing.
        clMyMutex.Release();
    }
}

//---------------------------------------------------------------------------
void App2Main(void* unused_)
{
    while (1) {
        // Claim the mutex.  This will prevent any other thread from claiming
        // this lock simulatenously.  As a result, the other thread has to
```

```
        // wait until we're done before it can do its work.  You will notice
        // that the Start/Done prints for the thread will come as a pair (i.e.
        // you won't see "Thread2: Start" then "Thread1: Start").

        clMyMutex.Claim();

        // Start our work (incrementing a counter).  Notice that the Start and
        // Done prints wind up as a pair when simuated with flAVR.

        KernelAware::Print("Thread2: Start\n");
        u32Counter++;
        while (u32Counter <= 1000000) {
            u32Counter++;
        }
        u32Counter = 0;
        KernelAware::Print("Thread2: Done\n");

        // Release the lock, allowing the other thread to do its thing.
        clMyMutex.Release();
    }
}
```

21.9 lab6_timers/main.cpp

This example demonstrates how to create and use software timers.

```
/*===========================================================================
     _____        _____        _____        _____
 ___|____ |__  __|____ |__  __|____ |__  __|____ |__
|    \ /  | | |    \    ||    |    || |/ /     ||___   |
|     \/  | | |     \   ||    \    || |/ \     ||___   |
|__/\__/|__|__||__|\__\  _||__|\__\  _||__|\__\  _||_____|
    |____|        |____|        |____|        |____|

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

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 6:  using Periodic and One-shot timers.

Lessons covered in this example include:

Takeaway:

===========================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP1_STACK_SIZE];
static void App1Main(void* unused_);

//---------------------------------------------------------------------------
static void PeriodicCallback(Thread* owner, void* pvData_);
static void OneShotCallback(Thread* owner, void* pvData_);

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);

    clApp1Thread.Start();
```

```cpp
    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
void PeriodicCallback(Thread* owner, void* pvData_)
{
    // Timer callback function used to post a semaphore.  Posting the semaphore
    // will wake up a thread that's pending on that semaphore.
    Semaphore* pclSem = (Semaphore*)pvData_;
    pclSem->Post();
}

//---------------------------------------------------------------------------
void OneShotCallback(Thread* owner, void* pvData_)
{
    KernelAware::Print("One-shot timer expired.\n");
}

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    Timer clMyTimer; // Periodic timer object
    Timer clOneShot; // One-shot timer object

    Semaphore clMySem; // Semaphore used to wake this thread

    // Initialize a binary semaphore (maximum value of one, initial value of
    // zero).
    clMySem.Init(0, 1);

    // Start a timer that triggers every 500ms that will call PeriodicCallback.
    // This timer simulates an external stimulus or event that would require
    // an action to be taken by this thread, but would be serviced by an
    // interrupt or other high-priority context.

    // PeriodicCallback will post the semaphore which wakes the thread
    // up to perform an action.  Here that action consists of a trivial message
    // print.
    clMyTimer.Start(true, 500, PeriodicCallback, (void*)&clMySem);

    // Set up a one-shot timer to print a message after 2.5 seconds, asynchronously
    // from the execution of this thread.
    clOneShot.Start(false, 2500, OneShotCallback, 0);

    while (1) {
        // Wait until the semaphore is posted from the timer expiry
        clMySem.Pend();

        // Take some action after the timer posts the semaphore to wake this
        // thread.
        KernelAware::Print("Thread Triggered.\n");
    }
}
```

21.10 lab7_events/main.cpp

This example demonstrates how to create and use event groups

```cpp
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__    __|_    |     |_    __|__  |__    _____
|    \  /  | |   |  \       ||       |     ||  |/ /     ||__   |
|     \/   | |   |   \      ||       |     || |/ /      ||__   |
|__/\__/|__|__|__|__|__\_\  __||__|__\_\   __||__|__\_\   __||_____|
     |_____|        |_____|        |_____|        |_____|

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

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 7: using Event Flags

Lessons covered in this example include:
-Using the EventFlag Class to synchronize thread execution
```

```
-Explore the behavior of the EVENT_FLAG_ANY and EVENT_FLAG_ALL, and the
 event-mask bitfield.

Takeaway:

Like Semaphores and Mutexes, EventFlag objects can be used to synchronize
the execution of threads in a system.  The EventFlag class allows for many
threads to share the same object, blocking on different event combinations.
This provides an efficient, robust way for threads to process asynchronous
system events that occur with a unified interface.

=========================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP1_STACK_SIZE];
static void App1Main(void* unused_);

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp2Thread;
static K_WORD awApp2Stack[APP2_STACK_SIZE];
static void App2Main(void* unused_);

//---------------------------------------------------------------------------
//
static EventFlag clFlags;

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);

    clApp1Thread.Start();
    clApp2Thread.Start();

    clFlags.Init();

    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    while (1) {
        uint16_t u16Flags;

        // Block this thread until any of the event flags have been set by
        // some outside force (here, we use Thread 2).  As an exercise to the
        // user, try playing around with the event mask to see the effect it
        // has on which events get processed.  Different threads can block on
        // different bitmasks - this allows events with different real-time
        // priorities to be handled in different threads, while still using
        // the same event-flag object.

        // Also note that EVENT_FLAG_ANY indicates that the thread will be
        // unblocked whenever any of the flags in the mask are selected.  If
        // you wanted to trigger an action that only takes place once multiple
        // bits are set, you could block the thread waiting for a specific
        // event bitmask with EVENT_FLAG_ALL specified.
        u16Flags = clFlags.Wait(0xFFFF, EVENT_FLAG_ANY);

        // Print a message indicaating which bit was set this time.
        switch (u16Flags) {
            case 0x0001: KernelAware::Print("Event1\n"); break;
```

```
            case 0x0002: KernelAware::Print("Event2\n"); break;
            case 0x0004: KernelAware::Print("Event3\n"); break;
            case 0x0008: KernelAware::Print("Event4\n"); break;
            case 0x0010: KernelAware::Print("Event5\n"); break;
            case 0x0020: KernelAware::Print("Event6\n"); break;
            case 0x0040: KernelAware::Print("Event7\n"); break;
            case 0x0080: KernelAware::Print("Event8\n"); break;
            case 0x0100: KernelAware::Print("Event9\n"); break;
            case 0x0200: KernelAware::Print("Event10\n"); break;
            case 0x0400: KernelAware::Print("Event11\n"); break;
            case 0x0800: KernelAware::Print("Event12\n"); break;
            case 0x1000: KernelAware::Print("Event13\n"); break;
            case 0x2000: KernelAware::Print("Event14\n"); break;
            case 0x4000: KernelAware::Print("Event15\n"); break;
            case 0x8000: KernelAware::Print("Event16\n"); break;
            default: break;
        }

        // Clear the event-flag that we just printed a message about.  This
        // will allow u16 to acknowledge further events in that bit in the future.
        clFlags.Clear(u16Flags);
    }
}

//---------------------------------------------------------------------------
void App2Main(void* unused_)
{
    uint16_t u16Flag = 1;
    while (1) {
        Thread::Sleep(100);

        // Event flags essentially map events to bits in a bitmap.  Here we
        // set one bit each 100ms.  In this loop, we cycle through bits 0-15
        // repeatedly.  Note that this will wake the other thread, which is
        // blocked, waiting for *any* of the flags in the bitmap to be set.
        clFlags.Set(u16Flag);

        // Bitshift the flag value to the left.  This will be the flag we set
        // the next time this thread runs through its loop.
        if (u16Flag != 0x8000) {
            u16Flag <<= 1;
        } else {
            u16Flag = 1;
        }
    }
}
```

21.11 lab8_messages/main.cpp

This example demonstrates how to pass data between threads using message passing.

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|_    |__  __|_    |__  _____
|    \  /  |  | ||   \   | |   |    | ||   |/ /     ||__    |
|     \/   |  | ||    \  | |   |    | ||   |\   \   ||__    |
|__/\__/ |__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
     |____|        |____|        |____|        |____|
--[Mark3 Realtime Platform]-------------------------------------------------

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

/*===========================================================================

Lab Example 8:  using messages for IPC.

In this example, we present a typical asynchronous producer/consumer pattern
using Mark3's message-driven IPC.


Lessons covered in this example include:
- use of Message and MessageQueue objects to send data between threads
- use of GlobalMessagePool to allocate and free message objects

Takeaway:

Unlike cases presented in previous examples that relied on semaphores or
event flags, messages carry substantial context, specified in its "code" and
```

```
"data" members.  This mechanism can be used to pass data between threads
extremely efficiently, with a simple and flexible API.  Any number of threads
can write to/block on a single message queue, which give this method of
IPC even more flexibility.

=========================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP1_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP1_STACK_SIZE];
static void App1Main(void* unused_);

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-point
// function used by the application thread.
#define APP2_STACK_SIZE (320 / sizeof(K_WORD))
static Thread clApp2Thread;
static K_WORD awApp2Stack[APP2_STACK_SIZE];
static void App2Main(void* unused_);

//---------------------------------------------------------------------------
static MessageQueue clMsgQ;

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);

    clApp1Thread.Start();
    clApp2Thread.Start();

    clMsgQ.Init();

    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    uint16_t u16Data = 0;
    while (1) {
        // This thread grabs a message from the global message pool, sets a
        // code-value and the message data pointer, then sends the message to
        // a message queue object.  Another thread (Thread2) is blocked, waiting
        // for a message to arrive in the queue.

        // Get the message object
        Message* pclMsg = GlobalMessagePool::Pop();

        // Set the message object's data (contrived in this example)
        pclMsg->SetCode(0x1337);
        u16Data++;
        pclMsg->SetData(&u16Data);

        // Send the message to the shared message queue
        clMsgQ.Send(pclMsg);

        // Wait before sending another message.
        Thread::Sleep(200);
    }
}

//---------------------------------------------------------------------------
void App2Main(void* unused_)
{
    while (1) {
        // This thread waits until it receives a message on the shared global
```

```
        // message queue.  When it gets the message, it prints out information
        // about the message's code and data, before returning the messaage object
        // back to the global message pool.  In a more practical application,
        // the user would typically use the code to tell the receiving thread
        // what kind of message was sent, and what type of data to expect in the
        // data field.

        // Wait for a message to arrive on the specified queue.  Note that once
        // this thread receives the message, it is "owned" by the thread, and
        // must be returned back to its source message pool when it is no longer
        // needed.
        Message* pclMsg = clMsgQ.Receive();

        // We received a message, now print out its information
        KernelAware::Print("Received Message\n");
        KernelAware::Trace(0, __LINE__, pclMsg->GetCode(), *((uint16_t*)pclMsg->
    GetData()));

        // Done with the message, return it back to the global message queue.
        GlobalMessagePool::Push(pclMsg);
    }
}
```

21.12   lab9_dynamic_threads/main.cpp

This example demonstrates how to create and destroy threads dynamically at runtime.

```
/*===========================================================================
      _____        _____        _____        _____
   ___|    _|__  __|__  |__    __|__  |__    __|__  |__   _____
  |    \  /    | ||     \       ||       |      ||    |/ /     ||___   |
  |     \/     | ||      \      ||       \      ||    |  <      ||__    |
  |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\   __||_____|
     |_____|        |_____|        |_____|        |_____|

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

Copyright (c) 2012-2016 Funkenstein Software Consulting, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
#include "memutil.h"

/*===========================================================================

Lab Example 9:  Dynamic Threading

Lessons covered in this example include:
- Creating, pausing, and destorying dynamically-created threads at runtime

Takeaway:

In addition to being able to specify a static set of threads during system
initialization, Mark3 gives the user the ability to create and manipu32ate
threads at runtime.  These threads can act as "temporary workers" that can
be activated when needed, without impacting the responsiveness of the rest
of the application.

===========================================================================*/
#if !KERNEL_USE_IDLE_FUNC
#error "This demo requires KERNEL_USE_IDLE_FUNC"
#endif
#if !KERNEL_USE_THREAD_CALLOUTS
#error "This demo requires KERNEL_USE_THREAD_CALLOUTS"
#endif
#if !KERNEL_TIMERS_TICKLESS
#error "This demo requires KERNEL_TIMERS_TICKLESS"
#endif

extern "C" {
void __cxa_pure_virtual(void)
{
}
}

//---------------------------------------------------------------------------
// This block declares the thread data for one main application thread.  It
// defines a thread object, stack (in word-array form), and the entry-poi   nt
// function used by the application thread.
#define APP1_STACK_SIZE (400 / sizeof(K_WORD))
static Thread clApp1Thread;
static K_WORD awApp1Stack[APP1_STACK_SIZE];
```

```
static void App1Main(void* unused_);

//---------------------------------------------------------------------------
// This block declares the thread stack data for a thread that we'll create
// dynamically.
#define APP2_STACK_SIZE (400 / sizeof(K_WORD))
static K_WORD awApp2Stack[APP2_STACK_SIZE];

#if KERNEL_USE_THREAD_CALLOUTS
#define MAX_THREADS (10)
static Thread*  apclActiveThreads[10];
static uint32_t au16ActiveTime[10];

static void PrintThreadSlack(void)
{
    KernelAware::Print("Stack Slack");
    for (uint8_t i = 0; i < MAX_THREADS; i++) {
        if (apclActiveThreads[i] != 0) {
            char szStr[10];

            uint16_t u16Slack = apclActiveThreads[i]->GetStackSlack();
            MemUtil::DecimalToHex((K_ADDR)apclActiveThreads[i], szStr);
            KernelAware::Print(szStr);
            KernelAware::Print(" ");
            MemUtil::DecimalToString(u16Slack, szStr);
            KernelAware::Print(szStr);
            KernelAware::Print("\n");
        }
    }
}

static void PrintCPUUsage(void)
{
    KernelAware::Print("Cpu usage\n");
    for (int i = 0; i < MAX_THREADS; i++) {
        if (apclActiveThreads[i] != 0) {
            KernelAware::Trace(0, __LINE__, (K_ADDR)apclActiveThreads[i],
      au16ActiveTime[i]);
        }
    }
}

static void ThreadCreateCallout(Thread* pclThread_)
{
    KernelAware::Print("TC\n");
    CS_ENTER();
    for (uint8_t i = 0; i < MAX_THREADS; i++) {
        if (apclActiveThreads[i] == 0) {
            apclActiveThreads[i] = pclThread_;
            break;
        }
    }
    CS_EXIT();

    PrintThreadSlack();
    PrintCPUUsage();
}

static void ThreadExitCallout(Thread* pclThread_)
{
    KernelAware::Print("TX\n");
    CS_ENTER();
    for (uint8_t i = 0; i < MAX_THREADS; i++) {
        if (apclActiveThreads[i] == pclThread_) {
            apclActiveThreads[i] = 0;
            au16ActiveTime[i]    = 0;
            break;
        }
    }
    CS_EXIT();

    PrintThreadSlack();
    PrintCPUUsage();
}

static void ThreadContextSwitchCallback(Thread* pclThread_)
{
    KernelAware::Print("CS\n");
    static uint16_t u16LastTick = 0;
    uint16_t        u16Ticks    = KernelTimer::Read();

    CS_ENTER();
    for (uint8_t i = 0; i < MAX_THREADS; i++) {
        if (apclActiveThreads[i] == pclThread_) {
            au16ActiveTime[i] += u16Ticks - u16LastTick;
            break;
        }
```

```cpp
    }
    CS_EXIT();

    u16LastTick = u16Ticks;
}

#endif

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();

    Kernel::SetThreadCreateCallout(ThreadCreateCallout);
    Kernel::SetThreadExitCallout(ThreadExitCallout);
    Kernel::SetThreadContextSwitchCallout(ThreadContextSwitchCallback)
      ;

    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp1Thread.Start();
    Kernel::Start();

    return 0;
}

//---------------------------------------------------------------------------
static void WorkerMain1(void* arg_)
{
    Semaphore* pclSem  = (Semaphore*)arg_;
    uint32_t   u32Count = 0;

    // Do some work.  Post a semaphore to notify the other thread that the
    // work has been completed.
    while (u32Count < 1000000) {
        u32Count++;
    }

    KernelAware::Print("Worker1 -- Done Work\n");
    pclSem->Post();

    // Work is completed, just spin now.  Let another thread destory u16.
    while (1) {
    }
}
//---------------------------------------------------------------------------
static void WorkerMain2(void* arg_)
{
    uint32_t u32Count = 0;
    while (u32Count < 1000000) {
        u32Count++;
    }

    KernelAware::Print("Worker2 -- Done Work\n");

    // A dynamic thread can self-terminate as well:
    Scheduler::GetCurrentThread()->Exit();
}

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    Thread    clMyThread;
    Semaphore clMySem;

    clMySem.Init(0, 1);
    while (1) {
        // Example 1 - create a worker thread at our current priority in order to
        // parallelize some work.
        clMyThread.Init(awApp2Stack, sizeof(awApp2Stack), 1, WorkerMain1, (void*)&clMySem);
        clMyThread.Start();

        // Do some work of our own in parallel, while the other thread works on its project.
        uint32_t u32Count = 0;
        while (u32Count < 100000) {
            u32Count++;
        }

        KernelAware::Print("Thread -- Done Work\n");

        PrintThreadSlack();

        // Wait for the other thread to finish its job.
        clMySem.Pend();

        // Once the thread has signalled u16, we can safely call "Exit" on the thread to
        // remove it from scheduling and recycle it later.
```

```
        clMyThread.Exit();

        // Spin the thread up again to do something else in parallel.  This time, the thread
        // will run completely asynchronously to this thread.
        clMyThread.Init(awApp2Stack, sizeof(awApp2Stack), 1, WorkerMain2, 0);
        clMyThread.Start();

        u32Count = 0;
        while (u32Count < 1000000) {
            u32Count++;
        }

        KernelAware::Print("Thread -- Done Work\n");

        // Check that we're sure the worker thread has terminated before we try running the
        // test loop again.
        while (clMyThread.GetState() != THREAD_STATE_EXIT) {
        }

        KernelAware::Print("  Test Done\n");
        Thread::Sleep(1000);
        PrintThreadSlack();
    }
}
```

Index